SEQUENTIAL EVOLUTIONARY OPERATIONS OF TRIGONOMETRIC SIMPLEX DESIGNS FOR HIGH-DIMENSIONAL UNCONSTRAINED OPTIMIZATION APPLICATIONS

Hassan Musafer

Under the Supervision of

Dr. Ausif Mahmood

DISSERTATION

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIRMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOHPY IN COMPUTER SCIENCE

AND ENGINEERING

THE SCHOOL OF ENGINEERING

UNIVERSITY OF BRIDGEPORT

CONNECTICUT

May, 2020

SEQUENTIAL EVOLUTIONARY OPERATIONS OF TRIGONOMETRIC SIMPLEX DESIGNS FOR HIGH-DIMENSIONAL UNCONSTRAINED OPTIMIZATION APPLICATIONS

Hassan Musafer

Under the Supervision of Dr. Ausif Mahmood

Approvals

Committee Members

Name	Signature	Date
Dr. Ausif Mahmood	Ausif Mahmood.	7-17-2020
Dr. Prabir Patra	P. Patra	9/25/20
Dr. Julius Dichter		9/3/2020
Dr. Miad Faezipour	Miad Faezipour	7/17/2020
Dr. Emre Tokgoz	Emre T.	7/17/2020

Ph.D. Program Coordinator

Dr. Khaled Elleithy — Khaled Elleithy — 9/28/2020

Chairman, Computer Science and Engineering Department

Dr. Ausif Mahmood — Ausif Mahmood. — 7-17-2020

Interim Dean, College of Engineering, Business, and Education

Dr. Khaled Elleithy — Khaled Elleithy — 9/28/2020

SEQUENTIAL EVOLUTIONARY OPERATIONS OF TRIGONOMETRIC SIMPLEX DESIGNS FOR HIGH-DIMENSIONAL UNCONSTRAINED OPTIMIZATION APPLICATIONS

SEQUENTIAL EVOLUTIONARY OPERATIONS OF TRIGONOMETRIC SIMPLEX DESIGNS FOR HIGH-DIMENSIONAL UNCONSTRAINED OPTIMIZATION APPLICATIONS

ABSTRACT

This dissertation proposes a novel mathematical model for the Amoeba or the Nelder-Mead simplex optimization (NM) algorithm. The proposed Hassan NM (HNM) algorithm allows components of the reflected vertex to adapt to different operations, by breaking down the complex structure of the simplex into multiple triangular simplexes that work sequentially to optimize the individual components of mathematical functions. When the next formed simplex is characterized by different operations, it gives the simplex similar reflections to that of the NM algorithm, but with rotation through an angle determined by the collection of nonisometric features. As a consequence, the generating sequence of triangular simplexes is guaranteed that not only they have different shapes, but also they have different directions, to search the complex landscape of mathematical problems and to perform better performance than the traditional hyperplanes simplex. To test reliability, efficiency, and robustness, the proposed algorithm is examined on three areas of large-scale optimization categories: systems of nonlinear equations, nonlinear least squares, and unconstrained minimization. The experimental results confirmed that the new algorithm delivered better performance than the traditional NM algorithm, represented by a famous Matlab function, known as "*fminsearch*".

In addition, the new trigonometric simplex design provides a platform for further development of reliable and robust sparse autoencoder software (SAE) for intrusion detection

system (IDS) applications. The proposed error function for the SAE is designed to make a trade-off between the latent state representation for more mature features and network regularization by applying the sparsity constraint in the output layer of the proposed SAE network. In addition, the hyperparameters of the SAE are tuned based on the HNM algorithm and were proved to give a better capability of extracting features in comparison with the existing developed algorithms. In fact, the proposed SAE can be used for not only network intrusion detection systems, but also other applications pertaining to deep learning, feature extraction, and pattern analysis. Results from experimental tests showed that the different layers of the enhanced SAE could efficiently adapt to various levels of learning hierarchy. Finally, additional tests demonstrated that the proposed IDS architecture could provide a more compact and effective immunity system for different types of network attacks with a significant detection accuracy of 99.63% and an F-measure of 0.996, on average, when penalizing sparsity constraint directly on the synaptic weights within the network.

To

My wonderful father and mother

My loving wife

My charming children, Ali, Ibrahim, and Elias

For their unconditional love, trust, and unending inspiration

ACKNOWLEDGEMENTS

I praise Allah who granted me the knowledge to complete this work successfully. This dissertation comes to its conclusion due to the assistance, guidance and trust of several people. I would like to thank all of them.

I owe my deepest sense of gratitude to my esteemed advisor Dr. Ausif Mahmood for his thoughtful guidance and valuable comments throughout the course of my dissertation.

I would like to offer my sincere thanks to Dr. Miad Faezipour, Dr. Prabir Patra, Dr. Julius Dicter, and Dr. Emre Tokgoz for serving on my supervisory committee, taking the time to evaluate this dissertation, and providing their valuable feedback and comments.

Most importantly, my deepest gratitude and sincere thanks are to my wonderful family for their understanding, encouragement, and inspiration. I am especially grateful to my parents, who always believe in me, for their love and trust. My grateful thanks are also to my loving and caring wife Elaf, and my three lovely children Ali, Ibrahim, and Elias for their love and everlasting inspiration. Finally, I would like to express my sincere thanks to my sisters and brothers and also my friends for their positive influence.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

Acc	Accuracy
ANMS	Adaptive Nelder Mead Simplex
ANN	Artificial Neural Network
CF	Cost Function
CNN	Convolutional Neural Network
Cs	Number of Classes
DBN	Deep Belief Network
DDoS	Distributed Denial-of-Service
DL-SVM	Deep Learning - Support Vector Machine
DMLP	Deep Multi-layer Perceptron
DoS	Denial-of-Service
DT	Decision Tree
E_F	Error Function
F_M	F-Measure
GNM	Genetic Nelder Mead
HNM	Hassan Nelder Mead
IDS	Intrusion Detection Systems
IoT	Internet of Things
IP	Internet Protocol
KNN	K-Nearest Neighbor

L_F	Loss Function
ML	Machine Learning
MLE-SVMs	Multi-layer Ensemble Support Vector Machines
MLP-PC	Multi-layer Perceptron-Payload Classifier
NM	Nelder Mead
RF	Random Forest
R_F	Regularization Function
SAE	Sparse Autoencoder
SGD	Stochastic Gradient Descent
SQL	Structured Query Language
SVM	Support Vector Machine
XSS	Cross-site Scripting

CHAPTER 1: INTRODUCTION

Unconstrained optimization algorithms are important for the rapid development of realistic and efficient systems with low costs. In addition, the models designed under these algorithms have made massive advances in the past two decades. In our opinion, it represents one of the most important fields for solving nonlinear optimization functions that arise in real world problems. The unconstrained optimization algorithms are widely used to choose various service levels, make short-term scheduling decisions, estimate system parameters, and solve similar statistical problems where the values of the functions are uncertain or prone to random error [1]. To study these algorithms, the best known techniques to solve unrestricted optimization are direct search methods that do not explicitly use derivatives as a necessary condition for obtaining the minimum of nonlinear functions [2]. Either because the calculations of exact first partial derivatives are either very expensive or or because it is time-consuming to use function evaluations to estimate derivatives. For example, a cost function can be defined with a complex computational structure that is very hard to find expressions for derivatives, or the output does not demonstrate derivatives [3, 4]. Another example illustrates that we cannot apply derivative-based methods directly if function values are noisy which incorporates an additive white-noise error [5]. For these reasons, direct search methods remain effective options for solving several types of unconstrained optimization problems when stochastic gradient techniques cannot be directly applied, or the solution may require great amount of computational effort [2].

1.1 Research Problem and Scope

One of the successful examples is the Nelder Mead simplex algorithm (1965) which outperforms most popular algorithms used for solving unconstrained optimization problems in the literature [1, 6, 7, 8, 9]. The famous Nelder and Mead algorithm was based on the previous work of Spendley et al. (1962), who devised the concept of simplex optimization [1]. Spendley et al. observed that a simplex of $(n+1)$ points or vertices is the minimal number of points required to form the initial simplex in n dimensional space [2]. Next, the function value is calculated at each of the vertices of the simplex in order to identify three specific points; the points associated with the highest, the second highest, and the lowest values of the objective function. The algorithmic iteration then allows to proceed towards a minimum by repeating a series of isometric reflections applied to the initial simplex, at the extreme point that has the highest function value (which is called worse point). All the simplex operations are performed along the line segment that is connecting the worse point and the center of gravity of the remaining n points [10]. If the cost function value does not replace the worst vertex with a better one then the algorithm performs a shrink operation around the best observed result. After scaling down the simplex to a smaller volume, the process of approaching to the optimum value continues until the appropriate coordinates are found at one of the vertices [11].

The contribution of the NM algorithm was to incorporate the simplex search with non-isometric reflections, designed to accelerate the search [6, 10, 12]. It was well-understood that the non-isometric reflections were designed to deform the simplex in a way that was better adapted to the features of mathematical functions [6, 13]. Nevertheless, when the number of variables under investigation increases, the NM simplex search often relies on reflection operations, which indicates that the exploration of the simplex search is similar to the method of Spendley et al. in high-dimensional problems. The procedures of the

traditional simplex search methods employed in high dimensional problems is not effective for landscape exploration in which case the additional features of Nelder and Mead are rarely used (1965). Torczon [14], proved the reason for inefficiency of the standard NM algorithm in high-dimensional problems to be the large number of reflection steps used as it approaches a minimum, particularly when $n > 2$. In addition, repeating of the non-isometric reflections to the initial simplex can cause in some cases the sequence of simplexes converges to a non-stationary point [15]. As a result, there are many modifications to the original NM algorithm in the literature. A good survey could be found in the resources [16, 17]. We classify the methods that successfully lead to enhancements of the Nelder-Mead algorithm into two types: permitting to test different step scales on the simplex for different operations [18, 19], and hybridizing with other techniques such as differential evolution [20], swarm [21] , and neural networks [22].

1.2 Motivation Behind the Research

The properties of the NM algorithm are quite simple to understand and develop a software solution, making it easy to get adopted in many fields of science and technology such as chemistry [23], engineering [24], biology [25], and medicine [26]. As a result, numerous modifications to the algorithm exist within the literature. To prove how efficient and reliable a modified NM algorithm, one key factor is the features of the test functions to which the algorithm is exposed. According to Moré et al. [27], testing the robustness and reliability of an unconstrained optimization algorithm in the literature is unaddressed because the algorithm has been tested on a small number of functions, and the starting points are close to the solution. The only way to evaluate an optimization algorithm is to test it on a collection of functions that have different structures and characteristics and belong to various optimization classes [28]. Moré et al. [27], present a large collection of test functions that is designed carefully for evaluating the reliability and robustness of

unconstrained optimization software. The collection covers systems of nonlinear equations, nonlinear least squares, and unconstrained optimization.

In addition, motivated by most of the developments that were contributed in proposing new variants of the NM algorithm, have been focused on controlling the step sizes of the simplex in different landscape. In our opinion, there is no real analysis about how efficient a solver of the NM algorithm relative to the other well-known solvers. This is the first attempt has been presented a new definition of the simplex optimization since the algorithm was developed in 1965, incorporating the non-isometric reflections with a rotation property to handle different landscape of mathematical problems. This research has diagnosed the problem of the NM algorithm in high dimension. There is nothing special about the NM algorithm, but the algorithm uses linear equations (reflection, expansion, contraction, and reduction) to solve non-linear equations. If we force the simplex to follow linear movements then the movements of the simplex are linear. This kind of approach might work well in low dimensional problems. However, using similar approach to handle high dimensional problems, particularly more than 10, is not useful. Because after few iterations the simplex becomes ineffective and some of the points come close to each others.

1.3 Contributions of the Proposed Research

In this work, we propose revolutionary operations of trigonometric simplex designs of the standard Nelder Mead simplex optimization algorithm (NM) (1965) for high-dimensional problems. Unlike the traditional hyperplanes simplex of the NM, the proposed simplex allows the components of the reflected vertex to fragment into multiple triangular simplexes and performs different operations of the proposed algorithm. Thus, the resulting solvers of triangular simplexes not only extract different non-isometric reflections of the proposed algorithm but also perform rotation through angles specified by the collection of features of the reflected vertex parameters in the hyperplane of the remaining vertices. The contri-

butions of this study include the following:

1. We propose a sequence of triangular simplex designs. Instead of launching one complicated simplex, we break it down into multiple triangular simplexes that work sequentially to optimize the individual components of mathematical problems.

2. We present a solid mathematical way to analyze the algorithm based on the vector theory, and to understand why the traditional NM algorithm fails to make further progress or gets trapped in local minima.

3. Based on our mathematical analysis, we incorporate the non-isometric reflections with a rotation property, allowing the reflected point to execute different operations of the proposed algorithm.

4. There is no reduction step in the proposed algorithm, we added two operations to the algorithm instead. The new two operations are reduction from good vertex to best vertex and reduction from worse vertex to best vertex.

5. We introduce a new sequence of operations, which turns out to be easier than the sequence of the standard NM.

6. We expect that the new properties of the algorithm make it appropriate for high-dimensional unconstrained optimization applications.

7. We propose a novel mathematical model for further development of robust, reliable, and efficient software for practical intrusion detection applications.

8. The proposed design provides a new platform for developing a compressed feature extraction based on imposing sparsity regularization on the weights, not the activations.

9. In addition, we use multiple sequences of trigonometric simplex designs as an optimizer to tune hyperparameters of the proposed sparse autoencoder that include number of nodes in the hidden layer, learning rate of the hidden layer, and learning rate of the output layer.

CHAPTER 2: BACKGROUND AND LITERATURE SURVEY

2.1 Introduction

The focus of this chapter is to survey the existing, techniques, software, and methods that were mainly developed based the NM algorithm. This chapter is organized as follows. Firstly, we define a framework to differentiate between similar algorithms and present two recently published papers related to our work. Next, we present a mathematical analysis of the NM algorithm based on the vector theory. Finally, we describe the theory of HNM and demonstrate how the algorithm manages to find a minimum.

2.2 Related Work

We demonstrate two recently improved NM algorithms regarding to the development of the algorithm. The purpose of doing that is to compare the developed algorithm to the state of the art algorithms and to validate experimental results accordingly. The algorithms use different mechanisms to measure efficiency and reliability: counting the number of function evaluations, and timing the algorithm. However, in our opinion, that is not enough to differentiate between similar algorithms. For example, it is important to know how many simplexes are being generated to reach an optimal value, and to know how the algorithm is capable of orienting the simplex towards the optima. That necessitates a comprehensive

framework that combines all the mentioned mechanisms together in order to obtain an objective comparison between similar algorithms.

A significant contribution to the NM algorithm was introduced by Gao and Han in 2012. They found a descent property effect on the step size which the new simplex has to make to replace the old one [29]. That explains why the standard NM is inefficient when the number of variables is more than two. The new implementation requires the expansion, contraction, and shrinkage operations of the NM algorithm to change adaptively while changing the dimensions to the optimization problems [30]. This makes the amount of simplex movement depend on the number of parameters being examined. The new algorithm, with the adaptive operations, outperforms the standard NM when the algorithm is tested on Moré et al. benchmark [29].

A modern contribution to the simplex optimization was done by Fejfar et al. (2017) [31], hybridizing NM with a genetic programming. The new research is considerable because it evolves NM genetically and produces deterministic simplexes that move biologically to locate an optima, rather than a stochastic genetic algorithm or a NM restricted to one simplex at a time. This type of hybridizing between the two methods is designed to increase the resolution of the genetic programming while looking for an optima because the NM algorithm seeks to find a local minima at high resolution. The authors claim that the shrinkage step of the standard NM could cause inconsistency because this is the only step that does not return a single simplex, and indeed it moves all vertices toward the best point. Therefore, they suggested that the reduction step includes exclusively the worst point and inner contraction is used to perform the job. The genetically enhanced NM turns out to be easier than the standard NM since it performs solely reflection, expansion, and inner contraction operations. However, outer contraction can be obtained by an iterative operation of inner contraction and reflection. In addition, the team decided to add one more vertex which is next to the second best. Since the new algorithm is population based, the center

of gravity depends on only three points: best, second best, and second worse vertices. On each iteration, three vertices are picked up to find the center of gravity and all the simplex movements have to be performed along the line segment that is linking the worse vertex and the center of gravity. The algorithm shows better accomplishment than the generic NM on Moré et al. dataset.

2.3 The Nelder-Mead Algorithm

In this section, we review the original NM algorithm (1965), as presented by [31, 32, 33, 34, 35, 36], which is slightly different than the version published by Nelder and Mead (1965). However, this change does not affect the basic concept of the NM, but illustrates the confusion that appeared in the algorithm. The NM algorithm is a derivative-free method that uses the concept of a simplex. The simplex is a geometric object defined by a formation of $n+1$ vertices $v_1, v_2, ..., v_{n+1}$ in $n-$dimensional space as follows.

$$det = \begin{bmatrix} v_1 & v_2 & \dots & v_{n+1} \\ 1 & 1 & \dots & 1 \end{bmatrix} \neq 0 \tag{2.1}$$

This condition ensures that the vertices are randomly distributed throughout the problem space. For example, when $n = 2$, the simplex is triangular, when $n = 3$, the simplex is a tetrahedron, and so on. Suppose that we need to specify the minimum of the function $f(x)$ that has n elements and without restrictions.

$$minf(x) \tag{2.2}$$

Where $f : R^n \rightarrow R$ is the cost function (CF), which is the problem that needs to be solved, and $x \in R^n$ is the parameter vector. To start the algorithm, we set up an initial simplex of $n+1$ vertices. A possible way to initialize a simplex, as suggested by [13], is to follow Pfefferś method. Based on the given starting point x_0 of dimension n, the initial

vertex is $v_1 = x_0$, and generate the reaming vertices as follows.

$$v_{i,j} = \begin{cases} v_{i-1,j} + \delta_u * v_{i-1,j} & \text{if } j = i-1 \text{ and } v_{i-1,j} \neq 0 \\ \delta_z & \text{if } j = i-1 \text{ and } v_{i-1,j} = 0 \\ v_{i-1,j} & \text{if } j \neq i-1 \end{cases} \tag{2.3}$$

The positive constant coefficients of zero term delta δ_z and usual delta δ_u are selected in a way that scales the initial simplex according to the characteristic lengths of the problem, the points $i = 2,3,...,n+1$, and the parameters $j = 1,2,...,n$. The cost function f is found at each extreme point (vertex) of the simplex, then the vertices are ordered with respect to ascending CF values so that v_1 is the best vertex and v_{n+1} is the worst vertex, $v_b = v_1, v_{sw} = v_n$, *and* $v_w = v_{n+1}$.

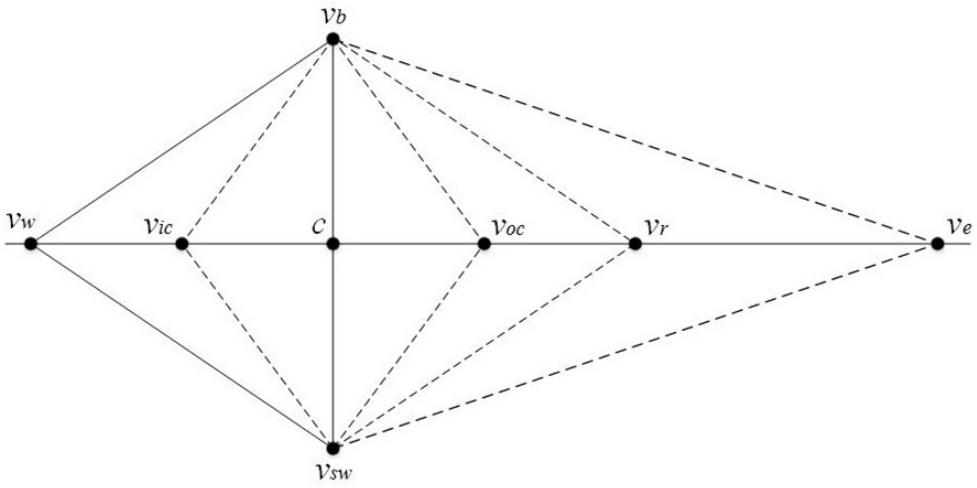

Figure 2.1: The geometrical analysis for an NM algorithm.

Next, we calculate the center of gravity c (centroid) of the simplex, which is the average of all the vertices except the worst point v_w.

$$c = \frac{1}{n} \sum_{i=1}^{n} v_i \tag{2.4}$$

After finding c, the worst point v_w, where $f(v_w)$ is largest, is reflected along the line segment connecting c and v_w, as seen in Figure 2.1, and replaced with a new vertex, by

implementing one of four different linear equations or operations: reflection, expansion, contraction, and reduction. In such operations, the same formula is applied to all these values, but with a different standard coefficient μ.

$$v_{new} = c + \mu(c - v_w) \tag{2.5}$$

$$v_r = c + \mu_r(c - v_w) = c + 1(c - v_w) \qquad (Reflection) \tag{2.6}$$

$$v_e = c + \mu_e(c - v_w) = c + 2(c - v_w) \qquad (Expansion) \tag{2.7}$$

$$v_{oc} = c + \mu_{oc}(c - v_w) = c + o.5(c - v_w) \qquad (Outer\ Contraction) \tag{2.8}$$

$$v_{ic} = c + \mu_{ic}(c - v_w) = c - o.5(c - v_w) \qquad (Inner\ Contraction) \tag{2.9}$$

Algorithm 1: The logical decisions for one iteration of the standard Nelder-Mead algorithm.

Sort the simplex's vertices descending, $f(v_1) \leq f(v_2) \leq ... \leq f(v_{n+1})$, $v_b = v_1, v_{sw} = v_n$, and $v_w = v_{n+1}$
Compute $f(v_r)$
if $f(v_r) < f(v_{sw})$ **then**
  Case1: (either Reflection or Expansion)
else
  Case2: (either Contraction or Shrinkage)
end if

Case1:
if $f(v_r) < f(v_b)$ **then**
Compute $f(v_e)$
  if $f(v_e) < f(v_r)$ **then**
    Replace v_w with v_e
  else
    Replace v_w with v_r
  end if
else
  Replace v_w with v_r
end if

Case2:
if $f(v_r) < f(v_w)$ **then**
Compute $f(v_{oc})$
  if $f(v_{oc}) < f(v_w)$ **then**
    Replace v_w with v_{oc}
  Compute $f(v_{ic})$
  else if $f(v_{ic}) < f(v_w)$ **then**
    Replace v_w with v_{ic}
  else
    Shrinkage
  end if
end if

If both the reflection and the contraction steps are rejected for not finding a better vertex, then the simplex is shrunk.

$$v_i = v_b - \mu_s \sum_{i=2}^{n+1} (v_b - v_i) = v_b - 0.5 \sum_{i=2}^{n+1} (v_b - v_i) \qquad (Reduction\ or\ Shrinkage) \qquad (2.10)$$

A new simplex is formed through replacing the worst vertex with a new one. The process of generating the sequence of simplexes, which might have different shapes, is continued, until the minimum point coordinates are found by one of the vertices of the simplex. The steps for how the NM method allocates a minimum point is given in Algorithm 1.

2.4 Hassan-Nelder-Mead Algorithm

We present in this section a theory of Hassan Nelder Mead (HNM) [37, 38], and describe the significance of the new, dynamic properties of the algorithm that make it appropriate for unconstrained optimization problems. The proposed simplex allows components of the next, generating vertex to be adaptive to different operations, based on the CF values. That is not the situation; as though, for the traditional NM algorithm, which forces the whole components of the simplex to execute a single operation such as expansion. When the resulting, next simplex is deformed by different operations, it gives the simplex similar reflections to that of the NM downhill algorithm, but with rotation through an angle determined by the collection of operations. As a consequence, the generating sequence of simplexes is guaranteed that not only they have different shapes, but also they have different directions, to search the complex landscapes of mathematical problems to perform better efficiency.

To initialize a simplex of HNM algorithm, an amendment to the Pfefferś method has to be considered in order to be consistent with the new features of the algorithm, as follows.

$$v_{i,j} = \begin{cases} v_{i-1,j} + \delta_u * v_{i-1,j} & \text{if } v_{i-1,j-1} \neq 0 \\ \delta_z & \text{if } v_{i-1,j-1} = 0 \end{cases} \qquad (2.11)$$

The values for usual delta δ_u is 0.05, for zero term delta δ_z is 0.00025, for points are

$i = 2, ..., 5$, and for elements are $j = 1, 2, ..., n$.

In general, HNM method consists of five vertices: a simplex (three vertices), a threshold, and a storage. Even though the simplex of HNM has only three vertices, it is capable of handling a mathematical function that has more than two dimensions. For example, in case of a two dimensional problem, both of the vertex and the point have two components; once we deal with a three dimensional problem, we need to add one component to each of the three vertices. In particular, the HNM algorithm uses multiple triangular simplexes (each simplex represents one component of a function), which work separately to optimize the individual components. Thus, the geometrical shape of the simplex is still a triangular, even we expanded the vertices in the example above with an additional component. That is to clarify why there is no need to expand our simplex further when dealing with high-dimensional problems (more than two). The other two vertices are needed from programing point of view. The need for a threshold is that when the algorithm reflects the worst point to find a better one, the new point will not replace the worst vertex. Instead, it will replace the current threshold, until all axial combinations of the vertices are carried out, to see if the new vertex is a good candidate to replace the worst vertex or not. While, the need for the storage is to maintain the existing threshold because there is no guarantee when the algorithm reflects the worst vertex, a better threshold can be found.

Before discussing how the HNM algorithm finds the coordinates of a minimum point, let us display a compact, mathematical way to analyze the HNM algorithm based on the vector theory, and understand why the original NM fails in some instances to make further improvements or gets trapped in local minimum. For example, suppose there is a function f that needs to find the minimum. The function $f(x, y)$ is calculated at the vertices, which are arranged ascending with respect to the CF values, such that.

$$v_b = v_1(x_1, y_1), v_{sw} = v_2(x_2, y_2), and\ v_w = v_3(x_3, y_3) \tag{2.12}$$

The construction process of the HNM algorithm uses three midpoints: M, S, C_1, as

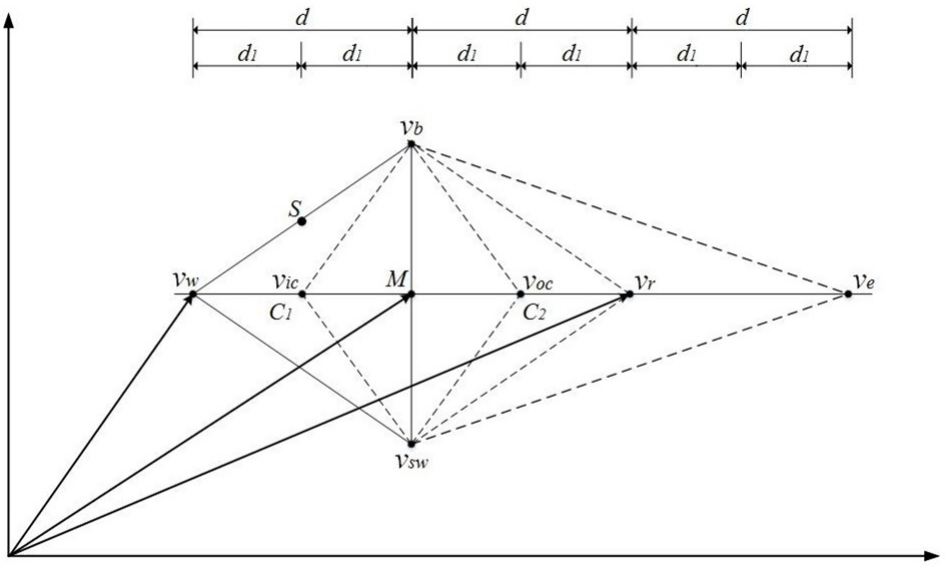

Figure 2.2: The geometrical analysis for an HNM algorithm based on vector theory.

seen in Figure 2.2, they can be found by calculating the average coordinates of the line segments connecting $(v_b \; and \; v_{sw})$, $(vb \; and \; v_w)$, and $(v_w \; and \; M)$ respectively. Hence to simplify the problem, our analysis depends on the combination of $x-$components and $y-$components (if there are more components we append the midpoints with additional axial components), to satisfy,

$$M(x_4, y_4) = \frac{v_b + v_{sw}}{2} = \left(x_4 = \frac{x_1 + x_2}{2}, y_4 = \frac{y_1 + y_2}{2} \right) \tag{2.13}$$

$$S(x_5, y_5) = \frac{v_b + v_w}{2} = \left(x_5 = \frac{x_1 + x_3}{2}, y_5 = \frac{y_1 + y_3}{2} \right) \tag{2.14}$$

$$C_1(x_6, y_6) = \frac{M + v_w}{2} = \left(x_6 = \frac{x_1 + x_3}{2}, y_6 = \frac{y_1 + y_3}{2} \right) \tag{2.15}$$

Note that to find the reflected point v_r, as it is possible from Figure 2, it can be achieved if we add the vectors M and d. The vector formula is below.

$$v_r = M + d = M + (M - v_w) = 2M - v_w = (2x_4 - x_3, 2y_4 - y_3) \tag{16}$$

A similar process could be used to find v_e, v_{ic}, and v_{oc}. The formulas are below.

$$v_e = M + 2d = M + 2(M - v_w) = 3M - 2v_w = (3x_4 - 2x_3, 3y_4 - 2y_3) \tag{2.16}$$

$$v_{ic} = C_1 = (x_6, y_6) \tag{2.17}$$

$$v_{oc} = C_2 = M + d_1 = M + (M - C_1) = 2M - C_1 = (2x_4 - x_6, 2y_4 - y_6) \tag{2.18}$$

Hence, HNM algorithm does not have a shrinkage step; instead, two operations are added to the algorithm: shrink from worse to best v_{rw} and shrink from good to best v_{rg}. The formulas are below.

$$v_{rg} = M = (x_4, y_4) \tag{2.19}$$

$$v_{rw} = S = (x_5, y_5) \tag{2.20}$$

Algorithm 2: The logic-based steps for a component of the HNM algorithm.

Sort the simplex's vertices, $f(v_1) \leq f(v_2) \leq ... \leq f(v_5)$ so that $v_b = v_1, v_{sw} = v_2, v_w = v_3$, $v_{th} = v_4$ and $v_{st} = v_5$
Compute $f(v_r)$
if $f(v_r) < f(v_{th})$ **then**
  Case1: (either Reflection or Expansion)
else
  Case2: (either Contraction or Shrinkage)
end if

Case1:	**Case2:**
Compute $f(v_e)$	**Compute** $f(v_{oc})$
if $f(v_e) < f(v_r)$ **then**	**if** $f(v_{oc}) < f(v_{th})$ **then**
Update v_{th} with v_e	Update v_{th} with v_{oc}
else	**else if** $f(v_{ic}) < f(v_{th})$
Update v_{th} with v_r	Update v_{th} with v_{ic}
end if	**else if** $f(v_{rg}) < f(v_{th})$
	Update v_{th} with v_{rg}
	else if $f(v_{rw}) < f(v_{th})$
	Update v_{th} with v_{rw}
	end if

It is noteworthy to mention that a combination of $x-$components of the HNM algorithm behaves exactly like a triangular simplex of the NM algorithm. The solution is

always implicit on the line connecting M and v_w. Now, if we consider two combinations or more, then the simplex as in the case of the HNM algorithm adapts to the various operations of the algorithm, so that the HNM algorithm's solution does not just reflect the opposite face of the simplex through the worse vertex, but rotate the reflected simplex through an angle, determined by the collection of operations. For example, suppose we need to find the minimum for the function $f(x,y)$. The solution of the NM algorithm may come to be reflection in x and y, whereas the solution of the HNM algorithm may come to be reflection in x but expansion in y. It can be a set of any two operations of the HNM algorithm. In fact, the HNM algorithm is designed to deform its simplex in a way that enables the algorithm to adapt to the high-dimensional features of mathematical problems, for better performance.Finally, the logical decisions of how the HNM allocate a minimum are explained in Algorithm 2.

CHAPTER 3: TESTING UNCONSTRAINED OPTIMIZATION APPLICATIONS

3.1 Introduction

In this chapter, we present the basic testing procedures that define the mechanisms to differentiate between similar algorithms, which are accuracy of the algorithm compared to the actual minima, timing the algorithm (in second), counting number of function evaluations, and counting number of simplex evaluations. Another significant factor is considered in this research, is the characteristics of the test functions that are exposed to an algorithm, to evaluate and distinguish between similar algorithms. According to Moré et al. [27], testing the reliability and robustness of an optimization algorithm has not been addressed in the literature because most of the testing procedures are small or the starting points are close to the solution. To address this need, Moré et al. have introduced a relatively large collection of different optimization functions, and designed guidelines for evaluating the reliability and robustness of unconstrained optimization software. The features of the test functions cover three areas: nonlinear least squares, unconstrained minimization, and systems of nonlinear equations. In addition, this chapter shows comparisons to two recently published algorithms, including adaptive NM and genetic NM algorithms.

3.2 Comparing HNM and ANMS on the Standard Function Set

A significant contribution to the NM algorithm was introduced by Gao and Han (2012). They proposed to change the standard coefficient μ values adaptively based on the dimension of mathematical problems n, and only for three operations: contraction, expansion, and shrinkage. The modified coefficient values are chosen as.

$$\mu_r = 1,\ \mu_e = 1 + \frac{1}{n},\ \mu_c = 0.75 - \frac{1}{2n},\ and\ \mu_s = 1 - \frac{1}{n} \tag{3.1}$$

The authors pointed out that when the diameter of the simplex is too small, the efficiency of contraction and expansion operations diminishes in high dimension. Based on a previous study introduced by Torczon [14], they proved mathematically that the inefficiency of the standard NM in high-dimensional problems is because the algorithm performs a larger number of reflection steps as it approaches a minimum, particularly $n > 2$. Therefore, the new implementation of the NM was proposed to reduce the probability of using large number of reflection steps and avoiding the rapid reduction in the simplex diameter. Thus, the well-scaled quadratic function, $f(x) = x^T.x$, with various dimension, $n = 2,4,...,100$, and starting vertex, $x_0 = [1,1,...,1]^T \in R^n$, were used to verify the effect of dimensionality on the standard NM and the adaptive NM. The stopping criteria should meet one of the following three points: $|f_i - f_1| \leq 10^{-8}$, where f_1 is the function value at the starting point, and $i = 2,3,...,10^6$, number of iterations $\leq 10^6$, and number of function evaluations $\leq 10^6$.

The results of the NM and adaptive NM algorithms are plotted in Figure 3.1, based on [29]. As shown from the figure part$-$A, the NM uses an excessive number of reflection steps, particularly when the algorithm moves to handle high dimension. Whereas, ANMS shows an extraordinary decrease and fluctuating response due to the adaptive step size of the simplex for different operations of the algorithm. In contrast with both, HNM

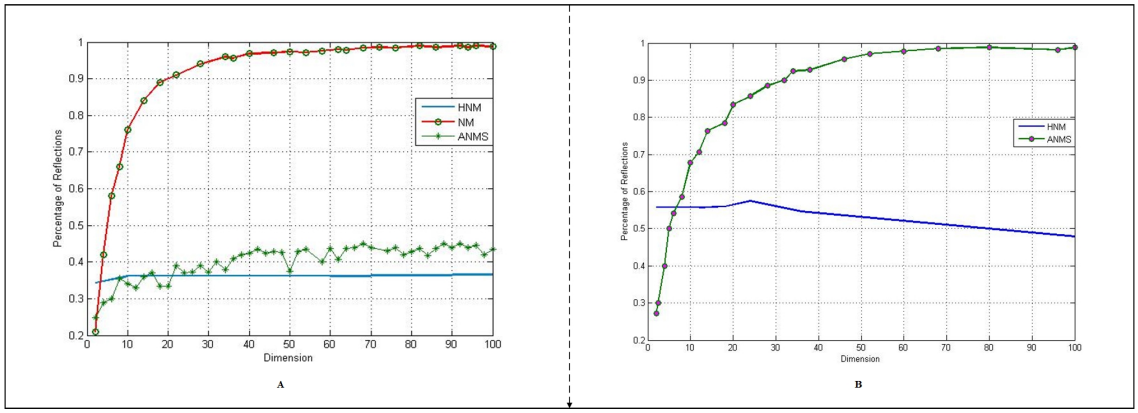

Figure 3.1: The percentage of the use of reflection steps relative to other processes for NM, ANMS, and HNM algorithms when testing on, A−quadratic function, B−Rosenbrock function.

appears a significant decrease of using reflection operations and a steady response along the dimension axis, confirming that the property of rotating the simplex through an angle follows the same patterns when exploring the same features of the mathematical functions − regardless of the number of dimension. Another careful study reveals that the results of the extended rosenbrock function shown in figure 3.1 part−B. The HNM algorithm uses significantly fewer number of reflection steps than the ANMS algorithm in high dimension. On the other hand, the simplex of the ANMS algorithm, even with the adaptive coefficient values μ, did not use the advantage of the expansion and contraction operations that have been introduced by Nelder and Mead to handle curved valleys function. It seems that the behavior of the simplex of the ANMS and Spendley algorithms are similar when handling rosenbrock function, particularly $n \geq 50$.

Another investigation is utilized to assess the efficiency of the HNM and the adaptive NM algorithms on badly-scaled convex functions, which has been modified by Gao and Han [29]. The function we consider to find the minimum is shown below.

$$min f(x) = x^T Dx + \sigma(x^T Bx)^2 \tag{3.2}$$

Where B and D are positive matrices that take the following forms.

$$D = diag([(1+\varepsilon),(1+\varepsilon)^2,...,(1+\varepsilon)^n]) \tag{3.3}$$

$$B = U^T U,\ U = \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix} \tag{3.4}$$

Where $\sigma \geq 0$ is a scalable parameter that determines the values of the matrix D, $\varepsilon \geq 0$ is a scalable parameter that controls the deviation of the quadratic portion of the function $f(x)$, and n is the number of dimension. The starting simplex is $[1,1,...,1]^T$. In the experimental test, the parameter values are chosen (ε,σ) = $(0,0)$, $(0,0.0001)$, $(0.05,0)$ and $(0.05,0.0001)$ respectively. The algorithm is terminated when the number of function evaluations exceed 10^6. The numerical results are summarized in Table 3.1.

Table 3.1: A performance evaluation of the HNM and Adaptive NM algorithms on the uniformly convex function.

(ε,σ,n)	ANMS *Accuracy* (*Function Evaluation*)	HNM *Accuracy* (*Function Eva.*) (*Simplex Eva.*) (*Time sec*)	Actual Minima
(0, 0, 10)	5.9143... 10^{-9} (898)	**2.1032**... **10^{-301}** (40465) (1228) (0.0468)	0.0
(0, 0, 20)	1.1343... 10^{-8} (2259)	**0.0** (89381) (1333) (0.2343)	0.0
(0, 0, 30)	1.5503... 10^{-8} (4072)	**0.0** (148198) (1427) (0.6249)	0.0
(0, 0, 40)	1.7631... 10^{-8} (7122)	**2.3075**... **10^{-212}** (131675) (954) (0.4531)	0.0
(0, 0, 50)	2.0894... 10^{-8} (9488)	**1.4049**... **10^{-251}** (186554) (1103) (0.7968)	0.0
(0, 0, 60)	3.5012... 10^{-8} (13754)	**2.9410**... **10^{-218}** (187109) (912) (0.9062)	0.0
(0, 0.0001, 10)	1.4603... 10^{-8} (1088)	**4.8700**... **10^{-301}** (39636) (1210) (0.1320)	0.0
(0, 0.0001, 20)	2.8482... 10^{-8} (4134)	**7.3553**... **10^{-282}** (76034) (1148) (0.4593)	0.0
(0, 0.0001, 30)	4.0639... 10^{-8} (13148)	**0.0** (148375) (1428) (2.1484)	0.0

(0, 0.0001, 40)	$9.3001\ldots\ 10^{-8}$	**7.1417**$\ldots$ **10^{-198}**	0.0
	(21195)	(105912) (810) (1.3428)	
(0, 0.0001, 50)	$8.1755\ldots\ 10^{-8}$	**6.0011**$\ldots$ **10^{-170}**	0.0
	(42403)	(112148) (696) (1.7601)	
(0, 0.0001, 60)	$1.0557\ldots\ 10^{-6}$	**4.9407**$\ldots$ **10^{-218}**	0.0
	(59626)	(183574) (902) (3.4754)	
(0.05, 0, 10)	$9.0552\ldots\ 10^{-9}$	**2.4269**$\ldots$ **10^{-211}**	0.0
	(910)	(27513) (850) (0.0781)	
(0.05, 0, 20)	$1.8433\ldots\ 10^{-8}$	**5.1226**$\ldots$ **10^{-236}**	0.0
	(2548)	(63077) (960) (0.3124)	
(0.05, 0, 30)	$2.6663\ldots\ 10^{-8}$	**0.0**	0.0
	(5067)	(136524) (1350) (1.2987)	
(0.05, 0, 40)	$3.6816\ldots\ 10^{-8}$	**2.9619**$\ldots$ **10^{-242}**	0.0
	(8598)	(125731) (968) (1.2329)	
(0.05, 0, 50)	$6.7157\ldots\ 10^{-8}$	**6.8064**$\ldots$ **10^{-247}**	0.0
	(13167)	(164956) (1018) (1.9201)	
(0.05, 0, 60)	$6.8945\ldots\ 10^{-8}$	**1.8043**$\ldots$ **10^{-180}**	0.0
	(20860)	(148225) (752) (2.0896)	
(0.05, 0.0001, 10)	$6.0454\ldots\ 10^{-9}$	**2.3857**$\ldots$ **10^{-211}**	0.0
	(994)	(28223) (865) (0.0937)	
(0.05, 0.0001, 20)	$1.5249\ldots\ 10^{-8}$	**4.9788**$\ldots$ **10^{-236}**	0.0
	(3788)	(63655) (967) (0.3749)	
(0.05, 0.0001, 30)	$4.0331\ldots\ 10^{-8}$	**0.0**	0.0
	(10251)	(135636) (1344) (1.5337)	
(0.05, 0.0001, 40)	$5.7407\ldots\ 10^{-8}$	**1.0577**$\ldots$ **10^{-242}**	0.0
	(18898)	(127958) (978) (1.4688)	
(0.05, 0.0001, 50)	$4.7431\ldots\ 10^{-7}$	**2.9410**$\ldots$ **10^{-218}**	0.0
	(37282)	(187109) (912) (0.9062)	
(0.05, 0.0001, 60)	$2.0786\ldots\ 10^{-7}$	**1.8043**$\ldots$ **10^{-180}**	0.0
	(61259)	(148225) (752) (2.6720)	

Let us take a comprehensive examination at the results of the proposed algorithm. The HNM generates a larger number of simplex evaluations, only if the algorithm could obtain a higher accuracy; and there is no effect observed of the dimension of the function on the response of the HNM algorithm. For instance, it is noticed that the simplex evaluations that are produced on the values $(0, 0, 20)$, $(0, 0, 30)$, $(0, 0.0001, 30)$, $(0.05, 0, 30)$ and $(0.05, 0.0001, 30)$, to reach the optimum, are the largest because we have achieved the highest accuracy on these values compared to the others. The lowest accuracy, however, always leads to fewer numbers of simplex evaluations like (0.05, 0.0001, 60). In fact, this indicates that the HNM algorithm is not sensitive to the number of parameters processed

when dealing with functions that have similar mathematical features. On the contrary, this is not the behavior of the ANMS algorithm, where the performance of the algorithm decreases when the dimension gets higher. Another important property of the HNM observed, is that the algorithm achieves a higher degree of accuracy for high dimension than for less. For example, see (0, 0, 60) compared to (0, 0, 40), (0, 0.0001, 60) compared to (0, 0.0001, 50), (0.05, 0, 50) compared to (0.05, 0, 10), and (0.05, 0.0001, 50) compared to (0.05, 0.0001, 10) because we do not have any control over the set of operations that should be selected by the algorithm to form the next simplex.

Table 3.2: Testing the HNM and adaptive NM algorithms on the standard Moré et al. function set.

Test Function (n)	Adaptive NM *Accuracy* (*Function Ev.*)	HNM *Accuracy* (*Function Ev.*) (*Simplex Ev.*) (*Time*)	Actual Minima
Brown badly scaled (2)	$2.0036\ldots\ 10^{-9}$ (275)	**0.0** (1512) (201) (0.0001)	0.0
Rosenbrock (2)	$8.1777\ldots\ 10^{-10}$ (159)	**0.0** (6963) (799) (0.0155)	0.0
Freudenstein and Roth (2)	$*$ $*$	**48.9842** (477) (56) (0.0001)	48.9842
Beale (2)	$1.3926\ldots\ 10^{-10}$ (107)	**0.0** (2418) (235) (0.0060)	0.0
Jennrich and Sampson (2)	$*$ $*$	**124.362** (295) (27) (0.0001)	124.362
Chebyquad (2)	$1.4277\ldots\ 10^{-8}$ (57)	**0.0** (1161) (148) (0.0050)	0.0
Powell badly scaled (2)	$1.4223\ldots\ 10^{-17}$ (700)	**0.0** (12530) (1249) (0.0155)	0.0
Helical valley (3)	$2.6665\ldots\ 10^{-4}$ (224)	**5.5715**$\ldots$ **10^{-29}** (32627) (2932) (0.0468)	0.0
Bard (3)	$*$ $*$	**8.2148**$\ldots$ **10^{-3}** (1882) (116) (0.0001)	$8.2148\ldots\ 10^{-3}$
Gaussian (3)	$1.2330\ldots\ 10^{-8}$ (70)	**1.1279**$\ldots$ **10^{-8}** (468) (42) (0.0155)	$1.1279\ldots\ 10^{-8}$
Meyer (3)	$*$ $*$	87.9484 (3766160) (357190) (10.2505)	87.9458
Box 3D (3)	$7.5589\ldots\ 10^{-2}$ (424)	**9.4096**$\ldots$ **10^{-29}** (516491) (51000) (0.5156)	0.0
Kowalik and Osborne (4)	$*$ $*$	**3.0750**$\ldots$ **10^{-4}** (15760) (1075) (0.0937)	$3.0750\ldots\ 10^{-4}$
Wood (4)	$9.1293\ldots\ 10^{-9}$ (711)	**8.3569**$\ldots$ **10^{-29}** (11637) (693) (0.0155)	0.0

Brown and Dennis (4)	**85822**	**85822.2**	85822.2
	(405)	(1594) (82) (0.0156)	
Powell singular (4)	$1.7814\ldots\ 10^{-7}$	**6.9169**$\ldots$ **10^{-323}**	0.0
	(353)	(61333) (4279) (0.0624)	
Extended Rosenbrock (4)	$8.1777\ldots\ 10^{-10}$	**0.0**	0.0
	(159)	(41754) (3149) (0.0312)	
Trigonometric (4)	$3.0282\ldots\ 10^{-4}$	**2.9714**$\ldots$ **10^{-202}**	0.0
	(197)	(9437) (760) (0.0156)	
Variably dimensioned (4)	$5.0684\ldots\ 10^{-9}$	**9.013**$\ldots$ **10^{-24}**	0.0
	(542)	(1856) (120) (0.0001)	
Brown almost linear (5)	$*$	**1.7990**$\ldots\ 10^{-28}$	0.0
	$*$	(17620) (900) (0.0155)	
Osborne 1 (5)	$*$	**5.854**$\ldots\ 10^{-5}$	$5.4648\ldots\ 10^{-5}$
	$*$	(1077857) (60000) (6.0612)	
Watson (6)	**2.2877**$\ldots$ **10^{-3}**	$2.2887\ldots\ 10^{-3}$	$2.2876\ldots\ 10^{-3}$
	(1846)	(1887646) (83813) (35.1947)	
Biggs EXP6 (6)	**5.5203**$\ldots$ **10^{-13}**	$9.5504\ldots\ 10^{-8}$	0.0
	(3923)	(1143461) (49000) (4.0632)	
Penalty I (4)	$2.2500\ldots\ 10^{-5}$	**2.2499**$\ldots$ **10^{-5}**	$2.2499\ldots\ 10^{-5}$
	(1436)	(608611) (39732) (0.6093)	
Penalty II (4)	$9.4755\ldots\ 10^{-6}$	**9.3762**$\ldots$ **10^{-6}**	$9.3762\ldots\ 10^{-6}$
	(197)	(28822754) (1877625) (48.84)	
Penalty II (10)	**2.9366**$\ldots$ **10^{-4}**	$2.9626\ldots\ 10^{-4}$	$2.9366\ldots\ 10^{-4}$
	(9741)	(1556688) (30000) (6.0156)	
Penalty I (10)	**7.0877**$\ldots$ **10^{-5}**	$7.6334\ldots\ 10^{-5}$	$7.0876\ldots\ 10^{-5}$
	(5410)	(716) (20) (0.0001)	
Discrete boundary (10)	$1.0388\ldots\ 10^{-7}$	**3.4543**$\ldots$ **10^{-26}**	0.0
	(1029)	(978531) (28400) (2.5400)	
Discrete boundary (20)	**3.1789**$\ldots$ **10^{-10}**	$7.1008\ldots\ 10^{-8}$	0.0
	(7535)	(1868130) (26200) (8.8801)	
Discrete boundary (30)	$3.0035\ldots\ 10^{-5}$	**9.9990**$\ldots$ **10^{-7}**	0.0
	(3860)	(2195921) (18878) (14.9849)	
Discrete boundary (40)	$1.6110\ldots\ 10^{-5}$	**9.9468**$\ldots$ **10^{-6}**	0.0
	(1029)	(14950) (65) (0.2030)	
Discrete boundary (50)	$8.8601\ldots\ 10^{-6}$	**5.7006**$\ldots$ **10^{-6}**	0.0
	(1905)	(11858) (40) (0.2187)	
Discrete boundary (60)	$5.3085\ldots\ 10^{-6}$	**3.5905**$\ldots$ **10^{-6}**	0.0
	(2125)	(7922) (22) (0.2187)	
Discrete integral (10)	$9.5926\ldots\ 10^{-9}$	**5.0031**$\ldots$ **10^{-30}**	0.0
	(774)	(6243) (167) (0.0468)	
Discrete integral (20)	$1.0826\ldots\ 10^{-8}$	**6.2216**$\ldots$ **10^{-23}**	0.0
	(3320)	(12686) (171) (0.3906)	
Discrete integral (30)	$2.1107\ldots\ 10^{-8}$	**8.6885**$\ldots$ **10^{-26}**	0.0
	(8711)	(18545) (167) (1.2969)	
Discrete integral (40)	$4.0741\ldots\ 10^{-8}$	**9.3356**$\ldots$ **10^{-20}**	0.0
	(18208)	(21829) (143) (2.8625)	
Discrete integral (50)	$4.7628\ldots\ 10^{-8}$	**2.2179**$\ldots$ **10^{-26}**	0.0
	(25961)	(28883) (152) (5.8127)	
Discrete integral (60)	$2.2644\ldots\ 10^{-8}$	**9.2152**$\ldots$ **10^{-15}**	0.0
	(38908)	(19539) (86) (5.7215)	

Broyden tridiagonal (10)	$2.5511\ldots 10^{-7}$ (740)	**5.8671**$\ldots$ **10**$^{\mathbf{-30}}$ (12707) (320) (0.0312)	0.0
Broyden tridiagonal (20)	$2.9158\ldots 10^{-7}$ (3352)	**9.0472**$\ldots$ **10**$^{\mathbf{-30}}$ (29731) (350) (0.0937)	0.0
Broyden tridiagonal (30)	$3.6927\ldots 10^{-7}$ (11343)	**8.2534**$\ldots$ **10**$^{\mathbf{-29}}$ (56107) (430) (0.2187)	0.0
Broyden tridiagonal (40)	$4.4076\ldots 10^{-7}$ (23173)	**4.8270**$\ldots$ **10**$^{\mathbf{-15}}$ (36714) (210) (0.2187)	0.0
Broyden tridiagonal (50)	$5.0978\ldots 10^{-7}$ (42013)	**3.9056**$\ldots$ **10**$^{\mathbf{-28}}$ (79970) (370) (0.5000)	0.0
Broyden tridiagonal (60)	$7.1834\ldots 10^{-7}$ (64369)	**4.9796**$\ldots$ **10**$^{\mathbf{-28}}$ (107131) (410) (0.7662)	0.0
Broyden banded (10)	$2.2149\ldots 10^{-7}$ (741)	**8.0296**$\ldots$ **10**$^{\mathbf{-31}}$ (11711) (291) (0.0468)	0.0
Broyden banded (20)	$5.1925\ldots 10^{-7}$ (1993)	**6.6213**$\ldots$ **10**$^{\mathbf{-31}}$ (35617) (453) (0.1875)	0.0
Broyden banded (30)	$6.4423\ldots 10^{-7}$ (3686)	**9.2296**$\ldots$ **10**$^{\mathbf{-30}}$ (50482) (426) (0.3947)	0.0
Broyden banded (40)	$1.0892\ldots 10^{-6}$ (6060)	**7.6307**$\ldots$ **10**$^{\mathbf{-30}}$ (76731) (480) (0.8380)	0.0
Broyden banded (50)	$1.2359\ldots 10^{-6}$ (8357)	**5.6685**$\ldots$ **10**$^{\mathbf{-30}}$ (96263) (480) (1.4372)	0.0
Broyden banded (60)	$1.0002\ldots 10^{-6}$ (10630)	**6.4886**$\ldots$ **10**$^{\mathbf{-27}}$ (95163) (400) (1.8118)	0.0
Trigonometric (6)	$1.8442\ldots 10^{-9}$ (437)	**4.3269**$\ldots$ **10**$^{\mathbf{-117}}$ (10618) (501) (0.0468)	0.0
Trigonometric (10)	$2.7952\ldots 10^{-5}$ (961)	**3.2619**$\ldots$ **10**$^{\mathbf{-263}}$ (42592) (1200) (0.2812)	0.0
Trigonometric (20)	$1.3504\ldots 10^{-6}$ (4194)	**8.8176**$\ldots$ **10**$^{\mathbf{-145}}$ (47431) (657) (2.0329)	0.0
Trigonometric (30)	$9.9102\ldots 10^{-7}$ (8202)	**4.7790**$\ldots$ **10**$^{\mathbf{-131}}$ (62440) (606) (3.2812)	0.0
Trigonometric (40)	$1.5598\ldots 10^{-6}$ (17674)	**6.5020**$\ldots$ **10**$^{\mathbf{-228}}$ (136943) (967) (24.6738)	0.0
Trigonometric (50)	$3.6577\ldots 10^{-7}$ (19426)	**4.5153**$\ldots$ **10**$^{\mathbf{-239}}$ (194040) (1063) (52.9998)	0.0
Trigonometric (60)	$9.6665\ldots 10^{-7}$ (31789)	**1.8885**$\ldots$ **10**$^{\mathbf{-152}}$ (119524) (611) (49.4998)	0.0
Extended Rosenbrock (6)	$1.3705\ldots 10^{-9}$ (1833)	**0.0** (19645) (799) (0.0312)	0.0
Extended Rosenbrock (8)	$*$ $*$	**0.0** (25986) (799) (0.0781)	0.0
Extended Rosenbrock (10)	$*$ $*$	**0.0** (32342) (799) (0.1093)	0.0
Extended Rosenbrock (12)	$3.3974\ldots 10^{-9}$ (10015)	**0.0** (38668) (799) (0.0780)	0.0
Extended Rosenbrock (24)	$4.2591\ldots 10^{-9}$ (50338)	**2.4299**$\ldots$ **10**$^{\mathbf{-27}}$ (1316486) (9650) (3.8439)	0.0
Extended Rosenbrock (30)	$5.4425\ldots 10^{-9}$ (156302)	**7.5132**$\ldots$ **10**$^{\mathbf{-10}}$ (5004910) (35000) (18.0038)	0.0

Extended Rosenbrock (36)	1.6616… 10^{-8}	**3.7628**… **10^{-28}**	0.0
	(119135)	(122109) (700) (0.5937)	
Extended Rosenbrock (100)	*	**3.7959**… **10^{-4}**	0.0
	*	(17851123) (35000) (210.1211)	
Osborne 2 (11)	*	**4.0137**… **10^{-2}**	4.0137… 10^{-2}
	*	(24701) (589) (1.6059)	
Extended Powell singular (12)	3.9417… 10^{-8}	**8.7476**… **10^{-304}**	0.0
	(25961)	(267683) (5459) (0.3437)	
Extended Powell singular (24)	4.8767… 10^{-7}	**5.6890**… **10^{-278}**	0.0
	(11156)	(517754) (5370) (1.2008)	
Extended Powell singular (40)	9.9115… 10^{-6}	**6.4638**… **10^{-219}**	0.0
	(38530)	(705169) (4310) (2.6560)	
Extended Powell singular (60)	1.9181… 10^{-5}	**9.8555**… **10^{-178}**	0.0
	(71258)	(804590) (3350) (4.4622)	
Extended Powell singular (100)	*	**6.5268**… **10^{-144}**	0.0
	*	(1293629) (3090) (12.9966)	
Variably dimensioned (6)	5.9536… 10^{-9}	**0.0**	0.0
	(1170)	(4173) (168) (0.0312)	
Variably dimensioned (12)	8.6227… 10^{-9}	**4.3026**… **10^{-27}**	0.0
	(4709)	(9198) (180) (0.0468)	
Extended Rosenbrock (18)	4.2290… 10^{-9}	**9.5323**… **10^{-23}**	0.0
	(29854)	(3424010) (45200) (7.6566)	
Variably dimensioned (24)	1.1237… 10^{-8}	**3.9643**… **10^{-19}**	0.0
	(35033)	(10419) (111) (0.0625)	
Variably dimensioned (30)	1.5981… 10^{-8}	**4.2881**… **10^{-24}**	0.0
	(67717)	(17190) (143) (0.1093)	
Variably dimensioned (36)	1.8116… 10^{-8}	**1.4958**… **10^{-10}**	0.0
	(209340)	(9782) (69) (0.0937)	
Linear−full rank (2)	*	**2.0000**	2.0
	*	(101) (16) (0.0039)	
Linear−full rank (6)	*	**2.0000**	2.0
	*	(329) (18) (0.0040)	
Linear−full rank (10)	*	**2.0000**	2.0
	*	(545) (18) (0.0155)	
Linear−rank 1 (2)	*	**0.2000**	0.2
	*	(194) (19) (0.0001)	
Linear−rank 1 (6)	*	**1.1538**	1.1538
	*	(888) (27) (0.0001)	
Linear−rank 1 (10)	*	**2.1428**	2.1428
	*	(1043) (20) (0.0155)	

To examine the performance of the ANMS and HNM algorithms on a wide range of optimization problems, numerical experiments are conducted on the standard test problems of Moré et al [28], and summarized in Table 3.2. As the experiments demonstrated, the results show the importance of those dynamical, selective operations are on the ability of

the proposed simplex to change its size and orientation and to obtain the best accuracy for most of the test problems. It has been also observed that the HNM algorithm was successful in following curved valleys functions like Rosenbrock. In addition, the test shows that the algorithm is able to generate the same number of simplex evaluations to reach the exact minimum for Rosenbrock (2, 6, 8, 10, and 12). While, for Rosenbrock (100), the behavior of the algorithm appeared a rapid decrease in the accuracy. For the systems of nonlinear equations, the performance of the HNM is noticed to satisfy the best observed results for Rosenbrock (2, 4, 6, 8, 10, 12, 24, 30, 36 and 100), Powell singular (4, 12, 18, 24, 30, 60, and 100), Powell badly scaled (2), Wood (4), Helical valley (3), Chebyquad (2), Brown almost linear (5), Discrete boundary (10, 20, 30, 40, 50 and 60), Discrete integral (10, 20, 30, 40, 50 and 60), Trigonometric (4, 6, 10, 20, 30, 40, 50, and 60), Variably Dimensioned (4, 6, 12, 18, 24, 30, and 36), Broyden tridiagonal (10, 20, 30, 40, 50, and 60), and Broyden banded (10, 20, 30, 40 50, and 60). Whereas, a few cases where ANMS algorithm successes to reach a higher solution: Watson (6) and Discrete boundary (20).

For the nonlinear least squares equations, we observe from this table that the HNM algorithm is always able to find a good approximation for the problems Linear−full rank (2, 6, and 10), Linear−rank 1 (2, 6, and 10), Rosenbrock (2, 4, 6, 8, 10, 12, 24, 30, 36 and 100), Powell singular (4, 12, 18, 24, 30, 60, and 100), Freudenstein and Roth (2), Bard (3), Kowalik and Osborne (4), Meyer (3), Box 3D (3), Jennrich and Sampson (2), Brown and Dennis (4), Osborne 1 (5), and Osborne 2 (11). For the unconstrained minimization, the HNM algorithm substantially outperforms the ANMS algorithm for this set of the functions: Helical valley (3), Powell badly scaled (2), Box three-dimensional (3), Variably dimensioned (4, 6, 12, 18, 24, 30, and 36), Gaussian (3), Penalty I (4), Penalty II (4), Brown badly scaled (2), Trigonometric (10, 50, and 60), Extended Rosenbrock (2, 4, 6, 8, 10, 12, 24, 30, 36 and 100), Extended Powell singular (4, 12, 18, 24, 30, 60, and 100), Beale (2), Wood (4), and Chebyquad (2). Overall, after testing the HNM algorithm

on a relatively large set of mathematical functions that belong to different optimization categories, it can be inferred that the algorithm is not tuned to particular functions. However, we note that the HNM algorithm seems to need significantly more function evaluations than the ANMS algorithm.

As it has known that the selective simplex of the HNM algorithm tries to elevate the dependency among function parameters, allowing the simplex to perform different operations of the HNM algorithm. However, in some kinds of optimization problems, because of the dependency it is found that a sequence of triangular simplexes achieves a better performance if we force the sequence to perform a linear movement (meaning all parameters of the reflected vertex have to perform one operation). It also affects what type of mathematical relationship is available such as linear or nonlinear (polynomial, exponential, logarithmic, or trigonometric). Although the selective simplex achieves high performance on well-scaled and bad scaled quadratic functions, it gets stuck and cannot proceed in some landscape of mathematical problems. Therefore, it was observed that to gain a higher accuracy on Moré et al. dataset, we need to force the simplex in some instances to follow one type of operations. This condition was rarely used and needed for Rosenbrock, Powell badly scaled, Beale, Chebyquad, Box 3D, Penalty I, Penalty II, Brown and Dennis, Powell singular, Wood, Watson, Broyden banded, Discrete boundary, and Broyden tridiagonal function. As a result, we allow the simplex to perform only one operation, but based on the logical expression of HNM when no better threshold can be found by the selective simplex.

Table 3.3: Percentage of Executions for Individual Operations of the HNM Algorithm.

Test Function (Dimension)	*Operations* (*% of Executions*)					
	V_R	V_E	V_{OC}	V_{IC}	V_{GB}	V_{WB}
Extended Rosenbrock (2−100)	47.90−57.59	12.90−16.26	1.71−3.18	20.99−24.25	1.11−4.70	0.88−5.48
Freudenstein and Roth (2)	40.59	22.77	3.96	32.67	0	0
Powell badly scaled (2)	58.40	17.10	0.96	32.79	0.33	0.37
Brown badly scaled (2)	37.57	11.11	5.90	39.13	2.79	2.48
Beale (2)	42.51	10.62	8.93	34.54	1.93	1.44
Jennrich and Sampson (2)	27.45	15.68	15.68	39.21	0	1.96
Chebyquad (2)	34.23	9.61	12.69	40.38	1.92	1.15
Helical valley (3)	62.81	15.16	1.63	18.53	0.78	1.05
Bard (3)	38.86	18.62	4.04	27.93	5.66	4.85
Gaussian (3)	41.34	0.96	8.65	45.19	1.92	1.92
Meyer (3)	73.50	11.40	1.55	12.04	0.65	0.83
Box 3D (3)	58.83	17.55	1.40	20.57	0.59	1.02
Wood (4)	49.82	11.87	4.97	27.95	2.18	3.19
Kowalik and Osborne (4)	54.25	15.97	2.64	23.24	1.67	2.19
Powell Singular (4−100)	43.30−49.66	11.99−12.44	4.27−5.41	29.66−32.09	2.29−3.07	1.71−3.89
Brown and Dennis (4)	35.42	16.14	7.17	34.97	3.58	2.69
Osborne 1 (5)	62.11	14.18	2.33	18.46	1.22	1.67
Osborne 2 (11)	50.51	18.25	3.10	23.96	1.92	2.23
Watson (6)	67.76	11.55	3.10	14.61	1.43	1.52
Biggs EXP6 (6)	50.91	18.19	2.63	24.73	1.42	2.09
Penalty I (4−10)	30.33−79.84	13.27−25.84	2.52−6.17	0.90−34.83	1.31−2.80	1.71−3.89
Penalty II (4−10)	39.86−95.02	2.15−17.08	1.64−5.27	0.93−28.51	0.21−3.84	0.02−5.41
Variably dimensioned (6−30)	38.20−41.22	7.98−11.21	7.95−10.84	29.89−32.24	5.56−6.84	3.28−5.88
Trigonometric (4−60)	40.79−66.61	0.80−4.51	0.97−7.66	22.67−55.16	0.47−3.70	0.53−2.34
Brown almost linear (5)	45.40	15.59	4.22	30.20	2.06	2.50
Discrete boundary (10−60)	55.54−61.10	7.52−18.68	0.12−2.08	19.66−30.86	0.12−1.21	0.25−1.68
Discrete integral (10−50)	42.17−43.84	6.59−11.08	7.33−10.60	30.48−33.38	4.32−5.45	2.91−3.36
Discrete Tridiagonal (10−60)	37.52−44.28	8.73−10.78	7.39−9.27	30.66−34.25	3.93−6.23	2.98−3.98
Broyden banded (10−60)	37.16−41.74	9.68−15.92	5.78−6.66	32.94−37.26	3.58−5.40	3.27−3.81
Linear−full rank (2−10)	25.00−27.77	27.77−31.25	0−0	43.75−44.44	0−0	0−0
Linear−rank 1 (2−10)	26.81−34.37	24.36−34.78	1.68−9.37	30.43−41.17	0−2.89	0−3.36

Han and Neumann have studied the effect of dimensioality on the standard NM when the objective function is strictly convex [12]. They found that the algorithm uses an infinite number of expansion and contraction operations and the diameter of the simplex converges to 0 as $n \longrightarrow \infty$. This complements the numerical experiments given by Torczon for the reason that the standard NM becomes inefficient in high dimension [14]. If this idea is now considered to investigate the percentage of use of different operations by a modified NM, one comes to the fact that a successful contribution to the algorithm can be done, if the modified one performs a moderate rate of reflection operations in order to benefit from the expansion and contraction operations introduced by Nelder and Mead, and a moderate rate of expansion and contraction operations in order to avoid the rapid reduction in the simplex diameter.

A more comprehensive examination on the importance of particular operations of the HNM algorithm is shown in Table 3.3. There are three operations were mainly executed for all testing functions, which are reflection, expansion and inside contraction. If the percentage of executions distributes among the three essential operations, then the algorithm has a great chance to allocate an optimal point within a moderate number of simplex evaluations. Whereas, if the rate of reflection operations is higher than 67 relative to the other operations, the algorithm uses an excessive number of function evaluations. That explains the poor performance of the algorithm when handling Watson, Penalty I, and Penalty II functions. It has also been noticed that the HNM algorithm performs a moderate rate of expansion and contraction operations relative to the others, reducing the chance of a rapid reduction in the simplex diameter, and improving the reliability and robustness of the HNM algorithm for large dimensional problems.

3.3 Comparing the Results of HNM with GNM

Fajfar et al. (2017) [31], hybridized Nelder Mead algorithm with genetic programming. The algorithm is population based simplexes that breed biologically over time to evolve deterministic simplexes, which ensure that they get closer to an optima with every new generation. The algorithm is designed to seek a global minima at high resolution. The genetic simplexes' programming structure is based on tree-syntax and is forming with a set of terminals and functions together the primitive set; that is important to maintain consistent type, function arguments and return values. The authors claim that the shrinkage step of the standard NM could cause inconsistency, because this is the only step that does not return a single vertex, and indeed it moves all vertices toward the best point. Therefore, they suggested that the reduction step includes exclusively the worst point, and inner contraction is used to perform the job. The genetically enhanced NM turns out to be easier than the standard NM since it performs solely reflection, expansion, and inner contraction operations. However, outer contraction can be obtained by an iterative operation of inner contraction and reflection. In addition, they have decided to add one more vertex to the set of the NM algorithm, which is the vertex with respect to the second best v_{sb}, so that the center of gravity of simplex is calculated for each new iteration and includes three vertices: best, second best, and second worse vertices. The new combination is: $v_b = v_1$ (best vertex), $v_{sb} = v_2$ (second best vertex), $v_{sw} = v_n$ (second worse vertex), and $v_w = v_{n+1}$ (worse vertex). The GNM algorithm extracts four vertices out of a population to find the center of gravity and all simplex movements is performed along the line segment connecting the worse vertex of the population and the computed centeriod on each iteration. The experiments have been run 20 times on 20 machines with a 2.66 GHz CPU Core i5; it took almost 12 hours to finish the test. Lastly, the algorithm converged successfully five times out of 20 to an acceptable solution (if the fitness obtained from a solver is lower than 10^{-5}).

Table 3.4: A comparison of HNM and GNM on Moré-Garbow-Hilstrom dataset.

Test Function (n)	GNM *Accuracy* (*Function Ev.*)	HNM *Accuracy* (*Function Ev.*) (*Simplex Ev.*) (*Time*)	Actual Minima
Rosenbrock (2)	$4.4373\ldots\ 10^{-31}$ (867)	**0.0** (6963) (799) (0.0155)	0.0
Freudenstein and Roth (2)	**48.9842** (425)	**48.9842** (477) (56) (0.0001)	48.9842
Powell badly scaled (2)	**0.0** (1957)	**0.0** (12530) (1249) (0.0155)	0.0
Brown badly scaled (2)	**0.0** (1949)	**0.0** (1512) (201) (0.0001)	0.0
Beale (2)	**0.0** (683)	**0.0** (2418) (235) (0.0060)	0.0
Jennrich and Sampson (2)	**124.362** (397)	**124.362** (295) (27) (0.0001)	124.362
Bard (3)	**8.2148**$\ldots$ $\mathbf{10^{-3}}$ (1067)	**8.2148**$\ldots$ $\mathbf{10^{-3}}$ (1882) (116) (0.0001)	$8.2148\ldots\ 10^{-3}$
Gaussian (3)	**1.1279**$\ldots$ $\mathbf{10^{-8}}$ (870)	**1.1279**$\ldots$ $\mathbf{10^{-8}}$ (468) (42) (0.0155)	$1.1279\ldots\ 10^{-8}$
Helical valley (3)	**0.0** (10679)	$5.5715\ldots\ 10^{-29}$ (32627) (2932) (0.0468)	0.0
Meyer (3)	**87.9458** (4511)	87.9484 (3766160) (357190) (10.2505)	87.9458
Box 3D (3)	**0.0** (2430)	$9.4096\ldots\ 10^{-29}$ (516491) (51000) (0.5156)	0.0
Powell singular (4)	$1.9509\ldots\ 10^{-61}$ (4871)	**6.9169**$\ldots$ $\mathbf{10^{-323}}$ (61333) (4279) (0.0624)	0.0
Wood (4)	**3.2183**$\ldots$ $\mathbf{10^{-30}}$ (2779)	$8.3569\ldots\ 10^{-29}$ (11637) (693) (0.0155)	0.0
Kowalik and Osborne (4)	**3.0750**$\ldots$ $\mathbf{10^{-4}}$ (1206)	**3.0750**$\ldots$ $\mathbf{10^{-4}}$ (15760) (1075) (0.0937)	$3.0750\ldots\ 10^{-4}$
Brown and Dennis (4)	**85822** (1288)	**85822.2** (1594) (82) (0.0156)	85822.2
Quadratic (4)	**0.0** (17173)	$2.7291\ldots\ 10^{-319}$ (17120) (1310) (0.0155)	0.0
Penalty I (4)	$3.9053\ldots\ 10^{-5}$ (289)	**2.2499**$\ldots$ $\mathbf{10^{-5}}$ (608611) (39732) (0.6093)	$2.2499\ldots\ 10^{-5}$
Penalty II (4)	**9.3762**$\ldots$ $\mathbf{10^{-6}}$ (5322)	**9.3762**$\ldots$ $\mathbf{10^{-6}}$ (28822754) (1877625) (48.84)	$9.3762\ldots\ 10^{-6}$
Osborne 1 (5)	**5.4648**$\ldots$ $\mathbf{10^{-5}}$ (2790)	$5.854\ldots\ 10^{-5}$ (1077857) (60000) (6.0612)	$5.4648\ldots\ 10^{-5}$
Brown almost linear (5)	**0.0** (2788)	$1.7990\ldots\ 10^{-28}$ (17620) (900) (0.0155)	0.0
Extended Rosenbrock (6)	$3.9443\ldots\ 10^{-31}$ (7494)	**0.0** (19645) (799) (0.0312)	0.0
Brown almost linear (7)	**5.3663**$\ldots$ $\mathbf{10^{-31}}$ (4104)	$8.6660\ldots\ 10^{-25}$ (69474) (2470) (0.0937)	0.0

Quadratic (8)	**0.0**	$5.3359\ldots 10^{-320}$	0.0
	(39785)	(37489) (1370) (0.0155)	
Extended Rosenbrock (8)	$2.7523\ldots 10^{-29}$	**0.0**	0.0
	(19164)	(25986) (799) (0.0781)	
Variably dimensioned (8)	**1.0292**$\ldots$ **10^{-31}**	$6.4049\ldots 10^{-24}$	0.0
	(7160)	(3989) (126) (0.0156)	
Extended Powell singular (8)	$9.7234\ldots 10^{-61}$	**6.9169**$\ldots$ **10^{-323}**	0.0
	(20353)	(167999) (5258) (0.2194)	
Watson (6)	**2.2877**$\ldots$ **10^{-3}**	$2.2887\ldots 10^{-3}$	$2.2876\ldots 10^{-3}$
	(5151)	(1887646) (83813) (35.1947)	
Extended Rosenbrock (10)	$9.0484\ldots 10^{-29}$	**0.0**	0.0
	(36268)	(32342) (799) (0.1093)	
Penalty I (10)	$9.4271\ldots 10^{-5}$	**7.6334**$\ldots$ **10^{-5}**	$7.0876\ldots 10^{-5}$
	(2100)	(716) (20) (0.0001)	
Penalty II (10)	$3.0000\ldots 10^{-4}$	**2.9626**$\ldots$ **10^{-4}**	$2.9366\ldots 10^{-4}$
	(1543)	(1556688) (30000) (6.0156)	
Trigonometric (10)	$2.7950\ldots 10^{-5}$	**3.2619**$\ldots$ **10^{-263}**	0.0
	(4252)	(42592) (1200) (0.2812)	
Osborne 2 (11)	**4.0137**$\ldots$ **10^{-2}**	**4.0137**$\ldots$ **10^{-2}**	$4.0137\ldots 10^{-2}$
	(7381)	(24701) (589) (1.6059)	
Extended Powell singular (12)	$5.7700\ldots 10^{-58}$	**8.7476**$\ldots$ **10^{-304}**	0.0
	(50117)	(267683) (5459) (0.3437)	
Quadratic (16)	**0.0**	**0.0**	0.0
	(112564)	(68660) (1260) (0.1120)	
Quadratic (24)	$8.0493\ldots 10^{-173}$	**5.0049**$\ldots$ **10^{-280}**	0.0
	(158849)	(99393) (1200) (0.2080)	

Table 3.4 summarizes the test results for both the HNM and the best solver of the GNM algorithms on Moré et al. dataset. The table also shows the dimension of the test functions n, the minimum values that the algorithms have been able to access, and the actual minima known for the functions. The minimum values written in bold indicate the highest accuracy or best observed outcomes at which an algorithm arrived. The results for quadratic functions are very interesting for both HNM and GNM; for low dimensions (n = 4 and 8), GNM was successful in reaching the actual minimum; while for high dimensions, n = 16, both HNM and GNH managed to obtain the exact minimum; and for n = 24, HNM succeeded in obtaining the best result, while GNM was stuck at some level and failed to make further improvement. However, HNM was observed to be the best in achieving the specified minimum or the higher accuracy for Rosenbrock (2, 6, 8, and 10) Powell singular (4, 8, and 12), Penalty I (4), Penalty I (10), Penalty II (10), and Trigonometric

(10). On the other hand, GNM was observed to have the best or exact results for Meyer (3), Box 3D (3), Osborne 1 (5), Brown almost linear (5 and 7), Watson (6), and variably dimensioned (8). Finally, in the results for higher-dimensional problems ($n \geq 10$), HNM was more successful than GNM in achieving the desired or exact solution. We can confirm that the HNM algorithm is designed to be computationally effective for higher dimensions and to have more control over the simplex than the GNM algorithm. It is now obvious how important these dynamically selected features are for HNM to achieve the best accuracy (n $\geq$ 10), compared with GNM.

Moré and Wild proposed normalized data profile for evaluating different solvers of derivative free optimization algorithms (2009), which measures the percentage of problems that can be solved with a particular number of simplex evaluations [39]. the normalized data profile formula is given below.

$$d_s(\alpha) = \frac{1}{\|P\|} size \left\{ p \in P : t_{p,s} \leq \alpha \right\} \tag{3.5}$$

where $\|P\|$ denotes the cardinality of P, p is the index of the number of functions $p \in P$, and $t_{p,s}$ is the performance metric for counting number of simplex evaluations.

The interesting idea about the normalized data profile is that it tests how fast (convergence rate) a solver relative to other solvers. It is also benefit from this performance measure to allocate a computational budget and to verify if a solver is tuned to particular mathematical problems.To measure the convergence rate for a function value within a certain number of simplex evaluations, we use the convergence test proposed by Moré and Wild (2009) [39]. The proposed convergence test is the following equation.

$$f(x_0) - f(x) \geq (1 - \tau)(f(x_0) - f_L) \tag{3.6}$$

Where $\tau > 0$ is a tolerance, x_0 is the starting point for the problem, and f_L is computed for each problem as the smallest obtained value of f.

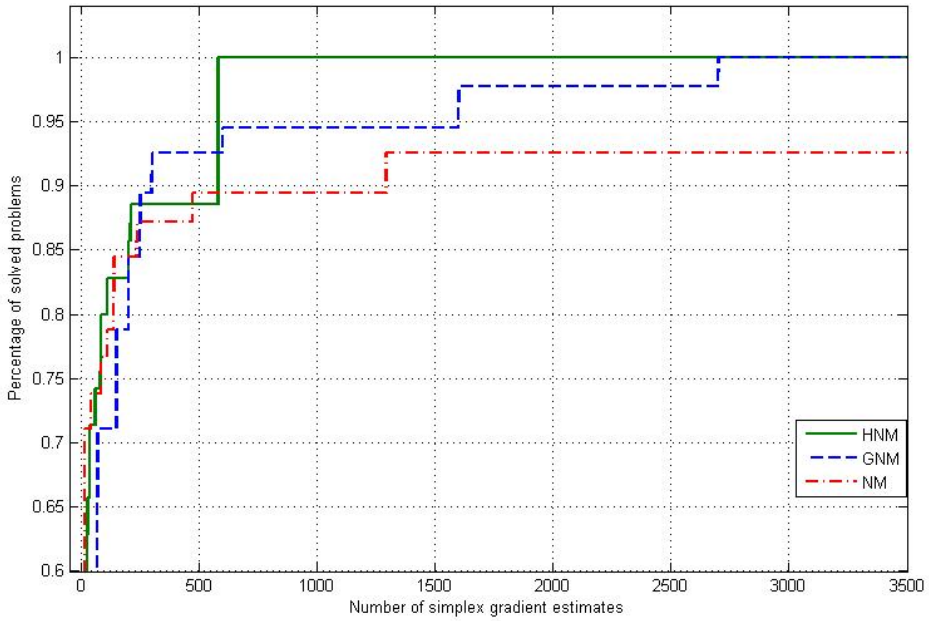

Figure 3.2: The percentage of solved test functions for the HNM, GNM, and NM algorithms at a precision of 10^{-3}.

Figure 3.2 shows the percentage of solved problems for the HNM, GNM, and NM algorithms at a precision of 10^{-3}. The results of the NM and GNM algorithms in the figure are based on [31]. As seen from the figure, the convergence rate of the NM algorithm is just slightly better than GNM for the first 250 simplex gradient estimates, but GNM catches up and appears a faster convergence rate approximately beyond 300 simplex evaluations. On the contrary, the HNM algorithm exhibits an overall better performance over both GNM and NM, and solves exactly 100% of the problems at 581 simplex evaluations. In fact, even the NM algorithm with the power of genetic programming is still constrained by the need to perform one operation for all simplex components. In addition, the normalized data profile measured for the GNM appears that the genetic simplex is tuned to particular functions and it took almost 2700 simplex evaluations to solve 100 percent of the function set. While, the normalized data profile measured for the NM algorithm requires more than 3500 simplex

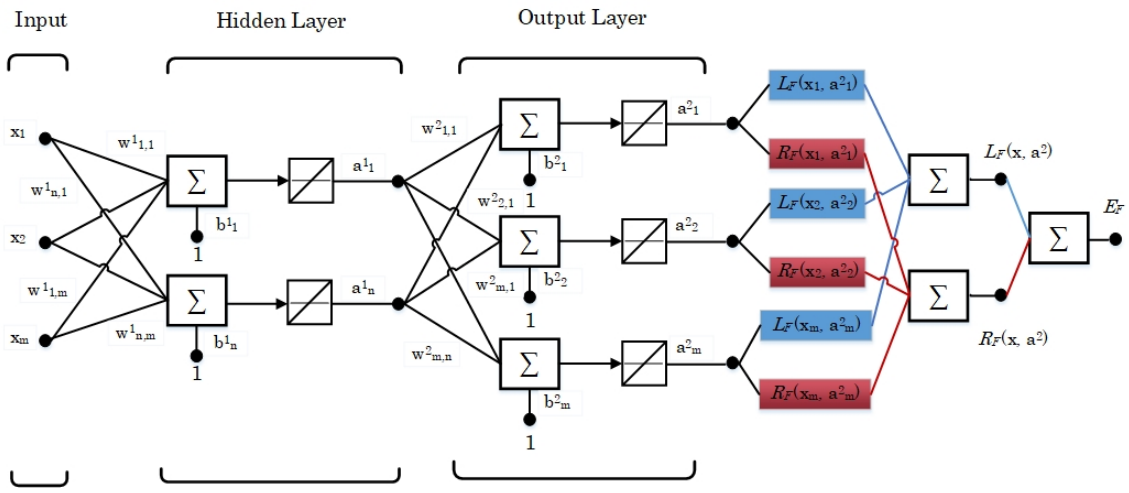

Figure 4.3: Example of sparse autoencoder approximation network.

For m input attributes and n hidden layer nodes, the equations that describe this operation are as follows.

$$a^1 = \begin{bmatrix} a^1_1 \\ \vdots \\ a^1_n \end{bmatrix} = \begin{bmatrix} w^1_{1,1} & w^1_{1,2} & \dots & w^1_{1,m} \\ \vdots & \vdots & \ddots & \vdots \\ w^1_{n,1} & w^1_{n,2} & \dots & w^1_{n,m} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} + \begin{bmatrix} b^1_1 \\ \vdots \\ b^1_n \end{bmatrix} \tag{4.2}$$

where $w^1 \in R^{n,m}$ is the weight matrix for the hidden layer and $b^1 \in R^{n,1}$ is the bias matrix associated with the hidden layer. As can be seen in Figure 4.3, the multilayer design of the SAE network has linear activation functions. Thus, the inputs of the output layer are purely represented by the vector $a^1 \in R^{n,1}$.

$$a^2 = \begin{bmatrix} a^2_1 \\ a^2_2 \\ \vdots \\ a^2_m \end{bmatrix} = \begin{bmatrix} w^2_{1,1} & \dots & w^2_{1,n} \\ \vdots & \ddots & \vdots \\ w^2_{m,1} & \dots & w^2_{m,n} \end{bmatrix} \begin{bmatrix} a^1_1 \\ \vdots \\ a^1_n \end{bmatrix} + \begin{bmatrix} b^2_1 \\ \vdots \\ b^2_m \end{bmatrix} \tag{4.3}$$

where $w^2 \in R^{m,n}$ and $b^2 \in R^{m,1}$ are the weight and bias matrices of the output layer. The

evaluations to solve the whole function set. This is evident when the GNM algorithm goes into high dimension; the algorithm needs additional function and simplex evaluations but has a slower convergence speed and less accuracy. See, for example, Extended Powell singular (12) and Quadratic (24).

CHAPTER 4: AN ENHANCED DESIGN OF SPARSE AUTOENCODER FOR NETWORK INTRUSION DETECTION SYSTEMS

4.1 Introduction

As a result of the increasing attacks on Internet-connected devices in the recent years, the study of Intrusion Detection Systems (IDS) has attracted strong interests from a wide range of different research communities, including information systems, security-software companies, and computer science fields [40]. An intrusion is defined as a set of actions that violates computer security policies, such as confidentiality, integrity and availability [41]. According to the annual report of the Asia Pacific Computer Emergency Response Team (CERT) published in 2018, newly emerging cyber-attacks and threats are evolving with modern technological advances such as artificial intelligence, deep learning, and new trends like the spreading of Internet of Things (IoT) devices. The main challenge is that attackers are highly skilled programmers and always keep novelty in their tools and techniques with the intent to exploit vulnerabilities in computer systems. Therefore, IDS has become an essential part of network security to monitor and respond to potential intrusions in any computing environment [42].

One of the popular defensive techniques for appropriate intrusion detection and prevention systems is based on machine learning (ML) and artificial neural network (ANN)

approaches. These ML - and ANN - based anti-threat systems generate a proactive response to stop attacks before they result in major security incidents [41, 43]. The advantage of using an ANN is to learn complex non-linear patterns in the input data. When IDS is implemented with ANNs and/or other machine learning algorithms, it provides a computer system with the great advantage of being capable of detecting intrusions and maintaining its security policies effectively [44]. Although IDS using ANN and ML methods are developed to provide optimum response in detecting intrusions, it is still difficult and challenging to detect all kinds of attacks in an efficient and high performance manner [42, 44].

The chapter is organized to include the following sections. Briefly glances at prior related work that were developed based on machine and deep learning techniques. The theory and mathematical model of the proposed architecture is presented and compared to the traditional sparse autoencoder. The results of the proposed idea applied on a well-known intrusion dataset is provided. A comparison with other related works is also provided with a detailed discussion for different performance metrics.

4.2 Related Work

This section briefly glances at intrusion detection algorithms related to the use of the CICIDS2017 dataset [45], which are developed mainly based on machine and deep learning techniques. In the literature, only a few IDS algorithms used sparse autoencoders to extract features based on latent representation concepts. The major challenge comes from the fact that high-level features produced by the traditional SAEs are designed to activate only a few number of the nodes in the hidden layers towards specific attributes of the input instances. This approach of extracting features fails to reflect the relationships of data instances by directly imposing a sparsity constraint in the hidden layers.

In [43, 46], the traditional SAE and support vector machine (SVM) have been used as feature extraction techniques while the random forest (RF) classifier was applied to de-

tect malicious attacks. The RF is an ensemble learning algorithm that combines bootstrap aggregation with random features selection to create a set of decision trees which result in a powerful prediction model with controlled variance [47]. In [48], the multilayer perceptron network and payload classifying algorithm (MLP-PC) was used to distinguish between network intrusions and benign traffic. The MLP network is a deep neural network that consists of five layers and utilizes Adam optimizer. The input layer is composed of 27 nodes, followed by three fully connected hidden layers. Each hidden layer is designed with 64 nodes, dropout probability 0.5, and rectified linear activation function. The output layer is a single node with a sigmoid activation function. This is while the payload classifier (PC) is a deep convolutional neural network (CNN) that consists of a character embedding layer, followed by four convolutional and pooling layers and two standard layers embedded with sigmoid function for classification. In [49], the Fisher Score algorithm (FS) was utilized for feature selection and the SVM, K-Nearest Neighbor (KNN) and Decision Tree (DT) algorithms were applied for intrusion detection, classifying two classes: DDoS or benign. The FS is a supervised feature selection algorithm that selects each feature independently according to a score measured by Fisher criterion [50]. In [51], a distributed model based on Spark was proposed using a collection of a deep belief network (DBN) and multi-layer ensemble support vector machines (MLE-SVMs). The DBN is a greedy layer-wise unsupervised learning model designed with a fine-tuning strategy to learn the relationships among low-level attributes and to represent a good set of hierarchical features. In [52], a deep learning based feature extraction technique and support vector machine (DL-SVM) were used to implement an effective and flexible IDS network. The authors in [53] presented the utilization of RF to keep the most effective features through a recursive feature elimination and deep multilayer perceptron (DMLP) structure to detect intrusion attacks.

4.3 Key Contributions

In spite of the successful contributions in the field of network intrusion detection using machine learning algorithms and deep networks to learn the boundaries between normal traffic and network attacks, it is still challenging to detect various attacks with high performance. In this research, we propose a novel mathematical model for further development of robust, reliable, and efficient software for practical intrusion detection applications. In this present work, we are concerned with optimal hyperparameters tuned for high performance sparse autoencoders for optimizing features and classifying normal and abnormal traffic patterns. The proposed framework allows the parameters of the back-propagation learning algorithm to be tuned with respect to the performance and architecture of the sparse autoencoder through a sequence of trigonometric simplex designs. These hyperparameters include the number of nodes in the hidden layer, learning rate of the hidden layer, and learning rate of the output layer. It is expected to achieve better results in extracting features and adapting to various levels of learning hierarchy as different layers of the autoencoder are characterized by different learning rates in the proposed framework.

The idea is viewed such that every learning rate of a hidden layer is a dimension in a multidimensional space. Hence, a vector of the adaptive learning rates is implemented for the multiple layers of the network to accelerate the processing time that is required for the network to learn the mapping towards a combination of enhanced features and the optimal synaptic weights in the multiple layers for a given problem. The suggested framework is tested on CICIDS2017; a reliable intrusion detection dataset that covers all the common, updated intrusions and cyber-attacks. Experimental results demonstrate that the proposed architecture for intrusion detection yields superior performance compared to recently published algorithms in terms of classification accuracy and F-Measure results.

4.4 Proposed Methodology

In this section, we present the proposed IDS architecture based on an enhanced SAE and RF algorithm, as depicted in Figure 4.1. The proposed IDS includes various modules for preprocessing huge amount of network packets, tuning the hyperparameters of SAE, and producing more mature and discriminating features. Typically, the preprocessing module identifies the minimum (Min) and maximum (Max) values of the basic features and normalizes them between 0 and 1. Moreover, features that have one value for different classes are eliminated. The hyperparameters tuning module selects five percent of the network packets, which will be used later as features for SAE's offline training to adjust the architecture of the SAE based on the HNM algorithm. The optimizing features module produces fewer number of features but more mature features and results in improved malicious attacks detection compared to traditional network features. The main modules of the proposed IDS are described in more detail hereafter.

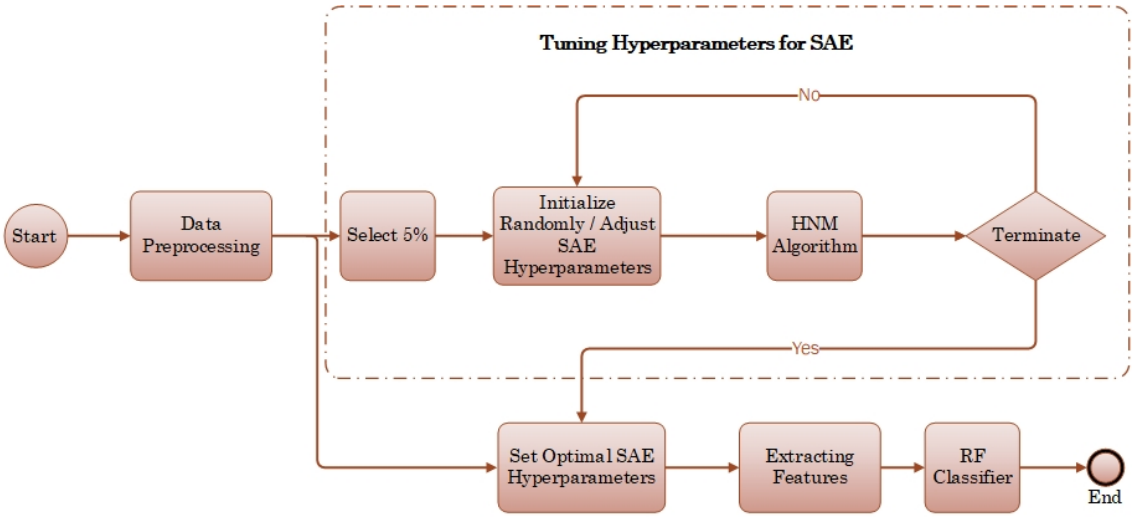

Figure 4.1: Architecture of the proposed IDS based on enhanced SAE and RF.

4.4.1 Data Preprocessing

The data prepossessing module breaks down the Internet Protocol (IP) and port number for sender and receiver, respectively, into four features instead of two; the CICIDS2017 dataset uses IP-port-sender and IP-port-receiver features. The benefit of doing so is that most intrusions follow a particular pattern for information gathering over the TCP/IP network. After that, the IP-sender and IP-receiver addresses are mapped to an integer representation. Finally, feature scaling is performed to ensure that all the data is in the same range between 0 and 1. Feature scaling is a unity-based normalization method and can be obtained by the following equation [54].

$$x_i = \frac{x_i - x_{min}}{x_{max} - x_{min}} \tag{4.1}$$

where x_{min}, x_{max} are the minimum and the maximum values for a particular feature x_i.

4.4.2 Hassan Nelder Mead Algorithm

In this work, we utilize the HNM algorithm [37] to tune the hyperparameters of SAE in order to mitigate the over-fitting problem raised in the hidden layer and to set optimal learning rates for different layers of the back-propagation learning algorithm. The HNM algorithm generates a sequence of trigonometric simplexes designed to extract different features of non-isometric reflections. Unlike the traditional hyperplanes simplex of the Nelder Mead (NM) algorithm [55, 56], the HNM simplex allows the components of the reflected vertex to fragment into multiple triangular simplexes and performs different operations of the algorithm. Thus, the resulting sequence of triangular simplexes not only extracts different non-isometric reflections, but also performs rotation through angles specified by the collection of features of the reflected vertex elements in the hyperplane of the remaining vertices. Therefore, the HNM algorithm is shown to be effective for unconstrained optimization problems. The detailed steps for one axial component of the HNM algorithm is

as follows:

Step 1. Initialize Triangular Simplex $(A, B, and\ C)$ and Threshold (Th), as shown in Figure 4.2:

A tetrahedron simplex is a geometrical object that has three vertices. Each vertex has n components, where n is the dimension of the mathematical problem. Since the HNM algorithm is employed to optimize three hyperparameters of SAE, n in our case equals to 3. Next, we sort the simplex vertices in a descending order according to an error function (E_F) that is defined later in the process to obtain four points associated with the lowest, second lowest, second highest, and highest E_F values, such that $A < B < C < Th$. Note, each of $(A, B, C, and\ Th)$ have three axial components (dimensions). The HNM algorithm optimizes a single component in each iteration, while pursuing to explore the curvatures of the E_F through six basic operations.

Step 2. Reflection D:

The HNM performs reflection along the line segment that is connecting the worst vertex C and the center of gravity, which is H to evaluate $E_F(D)$. The vector formula for D is given below.

$$D = A + B - C \tag{2}$$

Step 3. Expansion E:

If $(E_F(D) < E_F(Th))$, then the HNM executes expansion because it found a descent in that direction (see Figure 4.2). E is found by the following equation.

$$E = \frac{3A + 3B - 4C}{2} \tag{3}$$

a. If $(E_F(E) < E_F(D))$, then we replace the threshold point Th with E, and go to step 6.

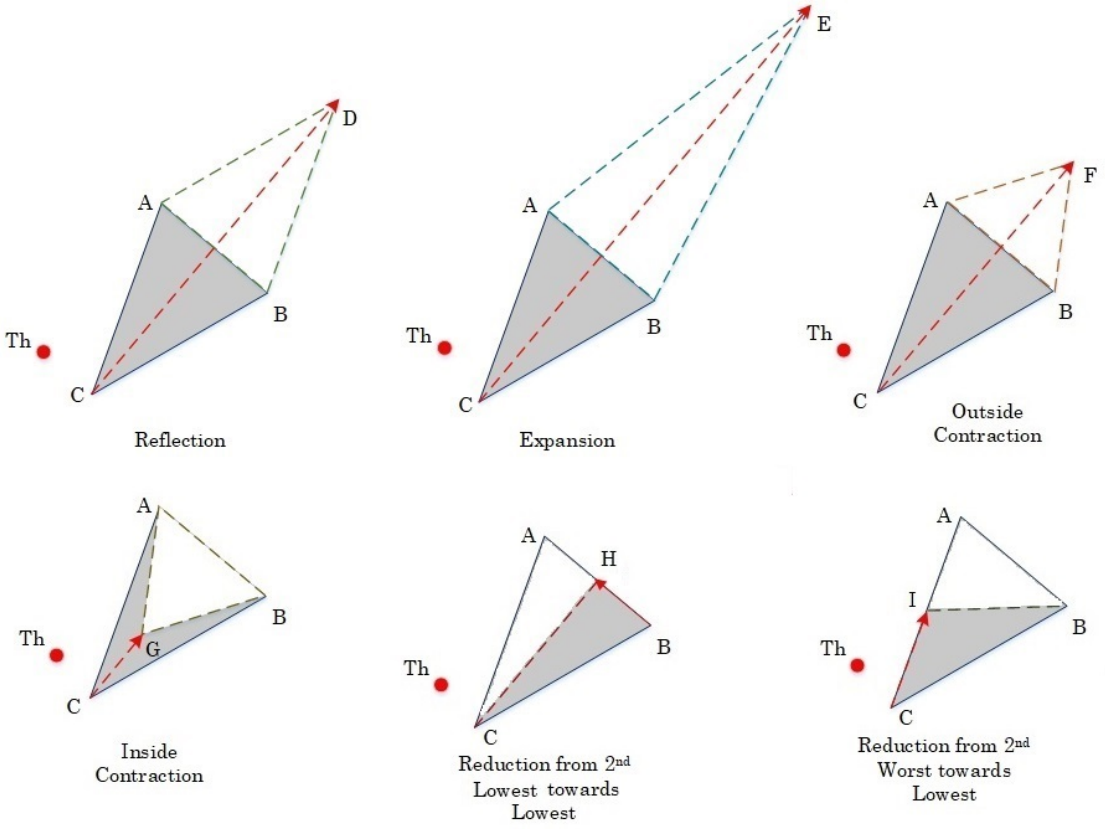

Figure 4.2: The basic operations of the HNM algorithm.

b. Otherwise Th is replaced by D, and the algorithm goes to step 6.

Step 4. Contraction F or G:

If $(E_F(D) > E_F(Th))$, then another point must be tested, which is F. If $(E_F(F) < E_F(Th))$, then F is kept and replaced with Th. If the condition of F is not met, then perhaps a better point is found somewhere between C and the centroid H. The point G is computed to see whether this point has a smaller function value than Th or not. The vector formulas for F and G are as follows.

$$F = \frac{3A + 3B - 2C}{4} \tag{4}$$

$$G = \frac{A + B + 2C}{4} \tag{5}$$

a. If either F or G have smaller E_F values than Th, then Th is updated and the algorithm goes to step 6.

b. Otherwise the algorithm moves to step 5.

Step 5. Reduction H or I:

The HNM algorithm performs two types of shrinkage operations. It shrinks the simplex either at the vertex that has the second lowest E_F value to evaluate H or at the second highest vertex to evaluate I. The HNM verifies the value of $E_F(H)$. If the condition of point H is not satisfied, then HNM shrinks the simplex along the line segment $\overline{AC}$ and evaluates $E_F(I)$. The HNM goes to step 6. The new vertices are given by:

$$H = \frac{A+B}{2} \tag{6}$$

$$I = \frac{A+C}{2} \tag{7}$$

Step 6. Termination Test:

The termination tests are problem-based and user-defined. In this work, the stopping criteria is primarily characterized by the designed error function of the SAE. It is encountered in examining the deviation of the error function from the true minimum by 10^{-4}, as indicated by the inequality below.

$$Stopping\ criterion = \sqrt{\sum_{i=1}^{N}(E_F(x_i,y_i,z_i) - \overline{E_F(x,y,z)})^2} < 10^{-4} \tag{8}$$

The termination criterion is evaluated for a predefined number of iterations (N). $(x, y, and\ z)$ correspond to the number of nodes in the hidden layer, learning rate of the hidden layer, and learning rate of the output layer, respectively.

If the condition of the termination test is satisfied, the HNM algorithm stops and returns the best architecture of SAE and the learning rates for the different layers of

the back-propagation algorithm. Otherwise, the algorithm sorts the simplex vertices and the Th and goes to step 2.

4.4.3 Proposed Sparse Autoencoder

Sparse autoencoder is an unsupervised learning algorithm whose training procedure involves a sparsity penalty, allowing only a few nodes of the hidden layers to activate when feeding a single sample into the network. The intuition behind this idea is that the algorithm is forced to sensitize a small number of individual nodes of the hidden layers towards specific features of the input sample [57, 58]. This form of regularization is accomplished by calculating the average activation nodes produced by the hidden layers over a collection of input samples. To satisfy the sparsity constraint, the mean computed over the training samples must be near 0 [57, 59]. The main problem, however, is that autoencoders often do not explicitly impose regularization on the weights of the network; instead, they regularize activations. As a result, poor performances are encountered with the early designs of sparse autoencoders such that sparsity makes it difficult for an autoencoder to approximate zero (or near zero) error loss function [59, 60].

In contrast to traditional autoencoders, this work proposes an alternative mathematical model for sparse autoencoders, which provides a new platform for developing a compressed feature extraction based on imposing sparsity regularization on the weights, not the activations. One solution to penalize weights within a network would be to impose regularization by the sparsity constraint in the output layer. As a result, the sparse autoencoder is encouraged to find a connection between the sparsity penalty and the learning to extract the latent features by selectively activating the number of variables (weights) of the network. The template of the proposed SAE is illustrated in Figure 4.3.

As discussed earlier, sparse autoencoder is an unsupervised learning algorithm and relies on conveying the outputs of one layer to become the inputs of the following layer.

outputs of the neurons in the last layer are considered as the SAE outputs, which are denoted by the vector $a^2 \in R^{m,1}$.

As shown in Figure 4.3, the proposed Error Function (E_F) is composed of two different parts. The first term is the Mean Squared Error or Loss Function (L_F) that measures the average squared difference between the estimated (output) and the actual (input) values. The second term is the proposed Regularization Function (R_F), which employs the sparsity constraint, mainly, for penalizing the weight matrices of the hidden and output layers. These terms are calculated as follows:

$$L_F = \frac{1}{m}\sum_{i=1}^{m}(x_i - a_i^2)^2 \tag{4.4}$$

$$R_F = \frac{1}{m}\sum_{i=1}^{m}\left((x_i+\rho)log\frac{x_i+\rho}{a_i^2+\rho} + (\rho - x_i)log\frac{\rho - x_i}{\rho - a_i^2}\right) \tag{4.5}$$

$$E_F = L_F + R_F \tag{4.6}$$

The KL divergence of the proposed R_F measures the difference between the probability distribution of the input and the output instances and it is more consistent with the loss function than the previous definition of SAE of NG [57]. Where ρ in Equation 4.5 is used to prevent the log function from getting the negative values and set equal to 10.

After propagating the input samples forward through the SAE network and obtaining the output vector (a^2), the next step is to evaluate the E_F from Equation 4.6. Since E_F is not an explicit function of the weights and bias in the SAE network, we need to specify a sensitivity measure that sensitizes the changes in E_F and propagates these changes backward through the network from the last layer to the first layer, in a process called the back-propagation learning algorithm.

To derive the recurrence relationship for the sensitivities, we use the Stochastic Gradient Descent algorithm (SGD) [61]. For the output layer, the SGD for updating the weight

and bias matrices can be expressed as follows.

$$w^2 = w^2 - \alpha^2 \frac{\partial E_F}{\partial w^2} \tag{4.7}$$

$$b^2 = b^2 - \alpha^2 \frac{\partial E_F}{\partial b^2} \tag{4.8}$$

where α^2 is the learning rate associated with output layer.

The only complication is that the E_F for a multilayer SAE design is an indirect function of the weights and bias. Thus, the chain rule theory is required to calculate the partial derivatives of E_F with respect to a third variable such as w or b in the hidden and output layers. By using the chain rule application, the derivatives of Equations 4.7 and 4.8 can be simplified to the following:

$$w^2 = w^2 - \alpha^2 \frac{\partial E_F}{\partial a^2} * \frac{\partial a^2}{\partial w^2} \tag{4.9}$$

$$b^2 = b^2 - \alpha^2 \frac{\partial E_F}{\partial a^2} * \frac{\partial a^2}{\partial b^2} \tag{4.10}$$

We denote the sensitivity at the output layer as s^2, which can be defined as:

$$s^2 = \frac{\partial E_F}{\partial a^2} \tag{4.11}$$

Then, Equations 4.9 and 4.10 become:

$$w^2 = w^2 - \alpha^2 \, s^2 \, (a^1)^T \tag{4.12}$$

$$b^2 = b^2 - \alpha^2 \, s^2 \tag{4.13}$$

where

$$s^2 = \begin{bmatrix} s_1^2 \\ s_2^2 \\ \vdots \\ s_m^2 \end{bmatrix} = \begin{bmatrix} \frac{-(x_1+10)}{log(10)\ (a_1^2+10)} + \frac{(10-x_1)}{log(10)\ (10-a_1^2)} - (x_1 - a_1^2) \\ \frac{-(x_2+10)}{log(10)\ (a_2^2+10)} + \frac{(10-x_2)}{log(10)\ (10-a_2^2)} - (x_2 - a_2^2) \\ \vdots \\ \frac{-(x_m+10)}{log(10)\ (a_m^2+10)} + \frac{(10-x_m)}{log(10)\ (10-a_m^2)} - (x_m - a_m^2) \end{bmatrix} \tag{4.14}$$

Following the same procedure for evaluating s^2, we can propagate the sensitivities backward from the output layer to the hidden layer as follows.

$$w^1 = w^1 - \alpha^1\ s^1\ (x)^T \tag{4.15}$$

$$b^1 = b^1 - \alpha^1\ s^1 \tag{4.16}$$

where α^1 is the learning rate associated with the hidden layer and s^1 is represented as follows.

$$s^1 = \begin{bmatrix} s_1^1 \\ \vdots \\ s_n^1 \end{bmatrix} = \begin{bmatrix} s_1^2 w_{1,1}^2 + s_2^2 w_{2,1}^2 + \cdots + s_m^2 w_{m,1}^2 \\ \vdots \\ s_1^2 w_{1,n}^2 + s_2^2 w_{2,n}^2 + \cdots + s_m^2 w_{m,n}^2 \end{bmatrix} \tag{4.17}$$

4.5 Experimental Results

As the rise in attacks on Internet connected-devices are being increased dramatically, it becomes significantly important to consider a reliable dataset that contains volumes of traffic diversity and covers a variety of attacks. Following this trend, we test our proposed IDS architecture on the CICIDS2017 dataset that covers almost the all common updated attacks such as DDoS, DoS, SQL Injection, Brute Force, XSS, Botnet, Infiltration, and Port Scan attacks. In addition, this section presents two experimental results in examining the efficiency and reliability of the proposed SAE network and shows comparisons with other

relevant works. While mitigating the effect of the over-fitting problem, we use the HNM algorithm to determine the number of nodes in the hidden layer based on the initial values of weights and bias in the network.

As shown in Figure 4.1, data preprocessing is the first step of preparing the records of the dataset, which includes unity-based normalization and eliminating the attributes that have one value in all instances of the dataset. After preprocessing, the volume of the dataset has been reduced to 70 features. Then, at least 5 percent of the reduced dataset is randomly selected to be used later by the HNM algorithm. The aim of using the HNM algorithm is to tune the hyperparameters of the SAE architecture, optimize the learning rates for the different layers, and set percentages of the sparsity for the different layers. Because the weights and bias values are initialized randomly, tuning the hyperparameters for the IDS design differs from one iteration to another. In this paper, we reported two experiments to observe how the hyperparameters are tuned based on random initialization and the results are summarized in Table 4.1. All the experiments and simulations were carried out using an Intel-Xeon processor with 3.70 GHz and 16 GB RAM, running Windows 10.

Table 4.1: Hyperparameters of the proposed SAE network.

Experiment	n	α^1	α^2	S_P^1	S_P^2	**Epoch**	**Time (seconds)**
1^{st}	12	0.01037246	0.1032246	22%	32%	180	4091
2^{nd}	5	0.00173640	0.0524296	22%	34%	126	2394

As illustrated in Table 4.1, different parametric measures are produced corresponding to the 1^{st} and 2^{nd} experiments. These hyperparameters include: number of nodes in the hidden layer (n), learning rate in the hidden layer (α^1), learning rate in the output layer (α^2), percentage of sparsity measured for the hidden layer (S_P^1), percentage of sparsity measured for the output layer (S_P^2), number of epochs (Epoch), and time in seconds (Time). Additionally, it can be seen that the values of the learning rates can be made to vary from one layer to another. This gives us a better features extraction strategy, where the different

layers can adapt to various levels of the learning hierarchy. This is while the percentages of sparsity, which are computed for the weight matrices, remain almost stable for both of the two conducted experiments.

The proposed SAE for IDS applications is implemented in C# language and tested on the CICIDS2017 dataset containing about 2,830,235 instances. After fine-tuning is performed using the HNM algorithm, the Random Forest (RF) classifier is used as the last layer to detect/distinguish fifteen classes, including the different types of attack packets and the normal traffic packets. Table 4.2 summarizes the test results in terms of F-Measure (F_M), and accuracy (Acc) to cover the experiments entirely. The Table also shows comparisons with our previous work and with some of the recently published algorithms. Three criteria are reported to characterize the different algorithms: number of classes (Cs), training set (Tr), and testing set (Ts). Finally, the feature extraction method and number of features are recorded from the corresponding papers in the last column of the Table. The (*) symbol indicates that the performance measure has not been reported.

Table 4.2: Summary of the experimental results and comparison with other techniques.

Algorithm	**Performance Measure**	**Feature Extraction**
1. MLP-PC [48]	$(F_M = 0.948)$ $(Acc = *)$	(27)
2. KNN [**?**]	$(F_M = 0.990)$ $(Acc = 99.0\%)$	FS (30)
3. MLE-SVMs [51]	$(F_M = 0.926)$ $(Acc = *)$	DBN (16)
4. DL-SVM [52]	$(F_M = 0.990)$ $(Acc = 97.8\%)$	(85)
5. DMLP [53]	$(F_M = *)$ $(Acc = 91.0\%)$	RF (10)
6. RF [43]	$(F_M = 0.995)$ $(Acc = 99.1\%)$	SAE (59)
7. **SAE-RF**	$(F_M =$ **0.996**$)$ $(Acc =$**99.63**$\%)$	**Proposed SAE (12)**
8. **SAE-RF**	$(F_M =$ **0.996**$)$ $(Acc =$**99.56**$\%)$	**Proposed SAE (5)**

4.6 Discussion

As demonstrated in Table 4.2, the two conducted experiments achieved results that outperform the existing solutions introduced for the updated and different types of network

attacks. Thereby, the proposed SAE architecture provides better performance to extract good set of features, which could reveal high levels of representation towards various characteristics of the latest intrusion attacks. This is proven by the test results. The features produced by the enhanced SAE technique had learned latent representation to sensitize the individual synaptic weights in the hidden layer and to generate keys for better classification accuracy and F-Measure results. The measurements of true positive rate, false positive rate, precision, recall, total number of epochs required to extract the latent features, and time in seconds (Time) for both experiments are summarized in Table 4.3. After tuning hyperparameters of the improved SAE, it required 3925 seconds to discover 12 latent features for the 1^{st} experiment and 2034 seconds to discover 5 latent features for the 2^{nd} experiment based on random initialization. Even though the second experiment took less time to represent the latent features, it failed to provide better performance in terms of the accuracy and false positive rate.

Table 4.3: Details of the results for two experiments.

Experiment	**True Positive Rate**	**False Positive Rate**	**Precision**	**Recall**	**Epoch**	**Time (sec.)**
1^{st}	0.996	0.009	0.996	0.996	76	3925
2^{nd}	0.996	0.011	0.996	0.996	27	2034

CHAPTER 5: CONCLUSION

In this work, we have proposed a Hassan-Nelder-Mead algorithm for multidimensional, unconstrained optimization applications. The proposed simplex has the ability to incorporate the non-isometric reflections of the NM with the rotation property, making it adaptive to skillfully explore the complex landscape of mathematical problems. After testing the HNM and adaptive NM algorithms on the uniformly and modified convex functions, HNM follows similar patterns when tracking similar characteristics of the mathematical functions − regardless of the dimensions of the functions. We have also studied the convergence speed toward the optima for the HNM and GNM algorithms with respect to the number of simplex gradient estimates. As stated in the experiment, HNM exhibits a faster convergence rate than the GNM algorithm. The statistical experiments have revealed that if the percentage of reflection steps is higher than 67, the HNM algorithm moves slowly toward the optima. Finally, we emphasize that the HNM algorithm is designed to be computationally effective for higher dimensions and to have more control over the simplex than the simplex of the NM algorithm.

In addition, we proposed an enhanced design of the SAE architecture for IDS applications. The proposed error function for the SAE is designed to make a trade-off between the latent state representation for more mature features and network regularization by applying the sparsity constraint in the output layer of the proposed SAE network. In addition, the hyperparameters of the SAE are tuned based on the HNM algorithm and were proved to give a better capability of extracting features in comparison with the existing developed al-

gorithms such as MLP-PC, MLE-SVMs, and DMLP. In fact, the proposed SAE can be used for not only network intrusion detection systems, but also other applications pertaining to feature extraction and pattern analysis. We emphasize that the successful contribution of allocating a set of optimal learning rates for different layers of the proposed SAE network has resulted in developing an efficient SAE architecture that can be used to discover latent features extraction. Results from experimental tests showed that the different layers of the enhanced SAE could efficiently adapt to various levels of the learning hierarchy. Finally, additional tests demonstrated that the proposed IDS architecture could provide a more compact and effective immunity system for different types of network attacks with a significant detection accuracy of 99.63% and an F-measure of 0.996, on average, when penalizing sparsity constraint directly on the synaptic weights within the network.

REFERENCES

[1] R. R. Barton and J. S. Ivey Jr, "Nelder-mead simplex modifications for simulation optimization," *Management Science*, vol. 42, no. 7, pp. 954–973, 1996.

[2] T. G. Kolda, R. M. Lewis, and V. Torczon, "Optimization by direct search: New perspectives on some classical and modern methods," *SIAM review*, vol. 45, no. 3, pp. 385–482, 2003.

[3] M. H. Wright, "Direct search methods: Once scorned, now respectable," *Pitman Research Notes in Mathematics Series*, pp. 191–208, 1996.

[4] C. Audet and W. Hare, *Derivative-free and blackbox optimization*. Springer, 2017.

[5] W. Swan, "Direct search methods, numerical methods for unconstrained optimization (ed. w. murray)," Academic Press, 1972.

[6] R. M. Lewis, V. Torczon, and M. W. Trosset, "Direct search methods: then and now," *Journal of computational and Applied Mathematics*, vol. 124, no. 1-2, pp. 191–207, 2000.

[7] M. H. Wright *et* al., "Nelder, mead, and the other simplex method," *Documenta Mathematica*, vol. 7, pp. 271–276, 2010.

[8] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, "Convergence properties of the nelder–mead simplex method in low dimensions," *SIAM Journal on optimization*, vol. 9, no. 1, pp. 112–147, 1998.

[9] A. Wouk *e*t al., *New computing environments: microcomputers in large-scale computing*, vol. 27, pp. 116–122. Siam, 1987.

[10] A. R. Conn, K. Scheinberg, and L. N. Vicente, *Introduction to derivative-free optimization*, vol. 8, pp. 141–161. Siam, 2009.

[11] W. Spendley, G. R. Hext, and F. R. Himsworth, "Sequential application of simplex designs in optimisation and evolutionary operation," *Technometrics*, vol. 4, no. 4, pp. 441–461, 1962.

[12] L. Han and M. Neumann, "Effect of dimensionality on the nelder–mead simplex method," *Optimization Methods and Software*, vol. 21, no. 1, pp. 1–16, 2006.

[13] M. Baudin, "Nelder mead user's manual.," Digiteo, 2009.

[14] V. J. Torczon, *Multidirectional search: a direct search algorithm for parallel machines*. PhD thesis, Rice University, 1989.

[15] K. I. McKinnon, "Convergence of the nelder–mead simplex method to a nonstationary point," *SIAM Journal on Optimization*, vol. 9, no. 1, pp. 148–158, 1998.

[16] J. C. Nash, *Compact numerical methods for computers: linear algebra and function minimisation*. CRC press, 1990.

[17] L. M. Rios and N. V. Sahinidis, "Derivative-free optimization: a review of algorithms and comparison of software implementations," *Journal of Global Optimization*, vol. 56, no. 3, pp. 1247–1293, 2013.

[18] R. O'Neill, "Corrigendum: Algorithm as 47: Function minimization using a simplex procedure," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 23, no. 2, pp. 252–252, 1974.

[19] C. Masters and H. Drucker, "Observations on direct search procedures," *IEEE Transactions on Systems Man and Cybernetics*, no. 2, p. 182, 1971.

[20] H. Lin, "Hybridizing differential evolution and nelder-mead simplex algorithm for global optimization," in *Computational Intelligence and Security (CIS), 2016 12th International Conference on*, pp. 198–202, IEEE, 2016.

[21] S.-K. S. Fan and E. Zahara, "A hybrid simplex search and particle swarm optimization for unconstrained optimization," *European Journal of Operational Research*, vol. 181, no. 2, pp. 527–548, 2007.

[22] E. A. Nawaz, T. N. Malik, E. N. Saleem, and E. E. Mustafa, "Globalized nelder mead trained artificial neural networks for short term load forecasting," *JB Appl Sci Res*, vol. 5, pp. 1–13, 2015.

[23] D. Wolf and R. Moros, "Estimating rate constants of heterogeneous catalytic reactions without supposition of rate determining surface steps—an application of a genetic algorithm," *Chemical Engineering Science*, vol. 52, no. 7, pp. 1189–1199, 1997.

[24] M. Camp and H. Garbe, "Parameter estimation of double exponential pulses (emp, uwb) with least squares and nelder mead algorithm," *IEEE transactions on electromagnetic compatibility*, vol. 46, no. 4, pp. 675–678, 2004.

[25] J. Yen, J. C. Liao, B. Lee, and D. Randolph, "A hybrid approach to modeling metabolic systems using a genetic algorithm and simplex method," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 28, no. 2, pp. 173–191, 1998.

[26] J. J. O'Hagan and A. Samani, "Measurement of the hyperelastic properties of 44 pathological ex vivo breast tissue samples," *Physics in Medicine & Biology*, vol. 54, no. 8, p. 2557, 2009.

[27] J. J. Moré, B. S. Garbow, and K. E. Hillstrom, "Testing unconstrained optimization software," *ACM Transactions on Mathematical Software (TOMS)*, vol. 7, no. 1, pp. 17–41, 1981.

[28] N. Andrei, "An unconstrained optimization test functions collection," *Adv. Model. Optim*, vol. 10, no. 1, pp. 147–161, 2008.

[29] F. Gao and L. Han, "Implementing the nelder-mead simplex algorithm with adaptive parameters," *Computational Optimization and Applications*, vol. 51, no. 1, pp. 259–277, 2012.

[30] N. Pham, A. Malinowski, and T. Bartczak, "Comparative study of derivative free optimization algorithms," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 592–600, 2011.

[31] I. Fajfar, J. Puhan, and Á. Bűrmen, "Evolving a nelder–mead algorithm for optimization with genetic programming," *Evolutionary computation*, vol. 25, no. 3, pp. 351–373, 2017.

[32] J. H. Mathews, K. D. Fink, *e*t al., *Numerical methods using MATLAB*, vol. 4. Pearson Prentice Hall Upper Saddle River, NJ, 2004.

[33] X.-S. Yang, *Engineering optimization: an introduction with metaheuristic applications*. John Wiley & Sons, 2010.

[34] E. K. Chong and S. H. Zak, *An introduction to optimization*, vol. 76. John Wiley & Sons, 2013.

[35] K. Price, R. M. Storn, and J. A. Lampinen, *Differential evolution: a practical approach to global optimization*. Springer Science & Business Media, 2006.

[36] J.-S. R. Jang, C.-T. Sun, and E. Mizutani, "Neuro-fuzzy and soft computing-a computational approach to learning and machine intelligence [book review]," *IEEE Transactions on automatic control*, vol. 42, no. 10, pp. 1482–1484, 1997.

[37] H. A. Musafer and A. Mahmood, "Dynamic hassan nelder mead with simplex free selectivity for unconstrained optimization," *Ieee Access*, vol. 6, pp. 39015–39026, 2018.

[38] H. Musafer, A. Abuzneid, M. Faezipour, and A. Mahmood, "An enhanced design of sparse autoencoder for latent features extraction based on trigonometric simplexes for network intrusion detection systems," *Electronics*, vol. 9, no. 2, p. 259, 2020.

[39] J. J. Moré and S. M. Wild, "Benchmarking derivative-free optimization algorithms," *SIAM Journal on Optimization*, vol. 20, no. 1, pp. 172–191, 2009.

[40] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, 2013.

[41] S. Anwar, J. Mohamad Zain, M. F. Zolkipli, Z. Inayat, S. Khan, B. Anthony, and V. Chang, "From intrusion detection to an intrusion response system: fundamentals, requirements, and future directions," *Algorithms*, vol. 10, no. 2, p. 39, 2017.

[42] M. Almseidin, M. Alzubi, S. Kovacs, and M. Alkasassbeh, "Evaluation of machine learning algorithms for intrusion detection system," in *2017 IEEE 15th International Symposium on Intelligent Systems and Informatics (SISY)*, pp. 000277–000282, IEEE, 2017.

[43] R. Abdulhammed, H. Musafer, A. Alessa, M. Faezipour, and A. Abuzneid, "Features dimensionality reduction approaches for machine learning based network intrusion detection," *Electronics*, vol. 8, no. 3, p. 322, 2019.

[44] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: an ensemble of autoencoders for online network intrusion detection," *arXiv preprint arXiv:1802.09089*, 2018.

[45] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization.," in *ICISSP*, pp. 108–116, 2018.

[46] R. Abdulhammed, M. Faezipour, H. Musafer, and A. Abuzneid, "Efficient network intrusion detection using pca-based dimensionality reduction of features," in *2019 International Symposium on Networks, Computers and Communications (ISNCC)*, pp. 1–6, IEEE, 2019.

[47] I. H. Sarker, A. Kayes, and P. Watters, "Effectiveness analysis of machine learning classification models for predicting personalized context-aware smartphone usage," *Journal of Big Data*, vol. 6, no. 1, p. 57, 2019.

[48] G. Watson, "A comparison of header and deep packet features when detecting network intrusions," tech. rep., 2018.

[49] D. Aksu, S. Üstebay, M. A. Aydin, and T. Atmaca, "Intrusion detection with comparative analysis of supervised learning techniques and fisher score feature selection algorithm," in *International Symposium on Computer and Information Sciences*, pp. 141–149, Springer, 2018.

[50] Q. Gu, Z. Li, and J. Han, "Generalized fisher score for feature selection," *arXiv preprint arXiv:1202.3725*, 2012.

[51] N. Marir, H. Wang, G. Feng, B. Li, and M. Jia, "Distributed abnormal behavior detection approach based on deep belief network and ensemble svm using spark," *IEEE Access*, vol. 6, pp. 59657–59671, 2018.

[52] D. Aksu and M. A. Aydin, "Detecting port scan attempts with comparative analysis of deep learning and support vector machine algorithms," in *2018 International Congress on Big Data, Deep Learning and Fighting Cyber Terrorism (IBIGDELFT)*, pp. 77–80, IEEE, 2018.

[53] S. Ustebay, Z. Turgut, and M. A. Aydin, "Intrusion detection system with recursive feature elimination by using random forest and deep learning classifier," in *2018 international congress on big data, deep learning and fighting cyber terrorism (IBIGDELFT)*, pp. 71–76, IEEE, 2018.

[54] K. Arai, R. Bhatia, and S. Kapoor, *Intelligent Computing: Proceedings of the 2019 Computing Conference*, vol. 1. Springer, 2019.

[55] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

[56] S. Albelwi and A. Mahmood, "A framework for designing the architectures of deep convolutional neural networks," *Entropy*, vol. 19, no. 6, p. 242, 2017.

[57] A. Ng *e*t al., "Sparse autoencoder," *CS294A Lecture notes*, vol. 72, no. 2011, pp. 1–19, 2011.

[58] J. Heaton, "Ian goodfellow, yoshua bengio, and aaron courville: Deep learning," Springer, 2018.

[59] A. Makhzani, *Unsupervised representation learning with autoencoders*. PhD thesis, University of Toronto, 2018.

[60] M. Ranzato, Y.-L. Boureau, and Y. L. Cun, "Sparse feature learning for deep belief networks," in *Advances in neural information processing systems*, pp. 1185–1192, 2008.

[61] H. B. Demuth, M. H. Beale, O. De Jess, and M. T. Hagan, *Neural network design*. Martin Hagan, 2014.

APPENDIX A: An Example of HNM in C# Language

An example of Hassan NM algorithm written in C# language. The cost function shown in the example is Rosenbrock function 2-dimension. The cost function is $CF = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$. The starting vertex is $(-1.2, 1)$.

We need to create Vertex Class as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace RosenBrockFunction
{
    class Vertix: ICloneable, IComparable
    {
            public Vertix() { }
            public Vertix(double a, double b, double cost)
            {
                this.a = a; // a = x1
                this.b = b; // b = x2
                this.cost = cost;
            }
```

```
double a;
public double A
{
    get { return a; }
    set { a = value; }
}
double b;
public double B
{
    get { return b; }
    set { b = value; }
}
double cost;
public double Cost
{
    get { return cost; }
    set { cost = value; }
}
public static Vertix TryRefluction( Vertix best, Vertix secondBest, Vertix worse,
Vertix updateRes, int i )     // Reflection
{
   Vertix res = new Vertix();
   switch (i)
   {
      case 0: res.a = (best.a + secondBest.a) - worse.a;
```

```
            res.b = updateRes.b;
            break;
        case 1: res.a = (best.b + secondBest.b) - worse.b;
            res.a = updateRes.a;
            break;
      }
     return res;
}
public static Vertix TryExpansion( Vertix best, Vertix secondBest, Vertix worse,
Vertix updateRes, int i )     // Expansion
{
   Vertix res = new Vertix();
   switch (i)
   {
      case 0: res.a = 3 * (best.a + secondBest.a) / 2 - 2 * worse.a;
            res.b = updateRes.b;
            break;
      case 1: res.a = 3 * (best.b + secondBest.b) / 2 - 2 * worse.b;
            res.a = updateRes.a;
            break;
      }
     return res;
}
public static Vertix ContractToC2( Vertix best, Vertix secondBest, Vertix worse,
Vertix updateRes, int i )     // Outside Contraction
{
```

```
    Vertex res = new Vertex();
    switch (i)
    {
       case 0: res.a = ((best.a + secondBest.a) / 2 - worse.a + best.a +
          secondBest.a) / 2;
          res.b = updateRes.b;
          break;
       case 1: res.a = ((best.b + secondBest.b) / 2 - worse.b + best.b +
          secondBest.b) / 2;
          res.a = updateRes.a;
          break;
     }
     return res;
}
public static Vertex ContractToC1( Vertex best, Vertex secondBest, Vertex worse,
Vertex updateRes, int i )     // Inside Contraction
{
    Vertex res = new Vertex();
    switch (i)
    {
       case 0: res.a = (((best.a + secondBest.a) / 2) + worse.a) / 2;
          res.b = updateRes.b;
          break;
       case 1: res.a = (((best.b + secondBest.b) / 2) + worse.b) / 2;
          res.a = updateRes.a;
          break;
```

```
    }
    return res;
}
public static Vertix ShrinkGToB( Vertix best, Vertix secondBest,
Vertix updateRes, int i )    // Reduction from a good vertex to a best vertex
{
   Vertix res = new Vertix();
   switch (i)
   {
      case 0: res.a = (best.a + secondBest.a) / 2;
         res.b = updateRes.b;
         break;
      case 1: res.a = (best.b + secondBest.b) / 2;
         res.a = updateRes.a;
         break;
    }
    return res;
}
public static Vertix ShrinkWToB( Vertix best, Vertix worse,
Vertix updateRes, int i )    // Reduction from a worse vertex to a best vertex
{
   Vertix res = new Vertix();
   switch (i)
   {
      case 0: res.a = (best.a + worse.a) / 2;
```

```
            res.b = updateRes.b;
            break;
        case 1: res.a = (best.b + worse.b) / 2;
            res.a = updateRes.a;
            break;
    }
    return res;
}
public override string ToString()
{
    return "Cost: " + cost.ToString() + "/nA: " + a.ToString();
}
public object Clone()
{
    return this.MemberwiseClone();
}
public int CompareTo( object obj)
{
  int ret = 0;
  if (obj is Vertix)
  {
                Vertix vr = (Vertix)obj;
                ret = this.cost.CompareTo(vr.cost);
  }
  return ret;
```

```
            }
        }
    }
```

Next, we need to crate a Simplex class, which is basically an array of the Vertex class as fellows.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace RosenBrockFunction
{
    class Simplex
    {
        Vertix[] = new Vertix[5]
        {
            new Vertix(-1.2, 1, 0),
            new Vertix(-1.2 + (-1.2 * 0.05 * 1), 1.05, 0),
            new Vertix(-1.2 + (-1.2 * 0.05 * 2), 1.10, 0),
            new Vertix(-1.2 + (-1.2 * 0.05 * 2), 1.10, 0),
            new Vertix(-1.2 + (-1.2 * 0.05 * 2), 1.10, 0),
        };
        public Vertix[] VArr
        {
            get { return vArr; }
```

```
    set { vArr = value; }
}
// To keep track about the different non-isometric features
string[] optimizeElementsInVertix = new string[2]
{
     "It did not optimize yet", "It did not optimize yet"
};
public string[] OptimizeElementsInVertix
{
    get { return optimizeElementsInVertix; }
    set { optimizeElementsInVertix = value; }
}
public Simplex() {}
public void ComputeCost (int i)
{
    this.vArr[i].Cost = 100 * Math.Pow( this.vArr[i].B -
    Math.Pow( this.vArr[i].A , 2), 2);
    this.vArr[i].Cost += Math.Pow(1 - this.vArr[i].A, 2);
}
public bool DoOptimizationOnVertixElements (int whichParameterOfVertix)
{
   bool optimize = false;
   double costTest = this.vArr[3].Cost;
   this.vArr[4] = (Vertix)this.vArr[3].Clone();
   this.vArr[3] = Vertix.TryRefluction( this.vArr[0], this.vArr[1], this.vArr[2],
```

```
this.vArr[3], whichParameterOfVertix);
ComputeCost(3);
if (this.vArr[3].Cost < costTest)     // Try Expansion
{
   this.vArr[4] = Vertix.TryExpansion( this.vArr[0], this.vArr[1], this.vArr[2],
   this.vArr[4], whichParameterOfVertix);
   ComputeCost(4);
   if (this.vArr[4].Cost < this.vArr[3].Cost)     // Expansion is a better than Ref.
   {
            optimize = true;
            optimizeElementsInVertix[whichParameterOfVertix] =
            "Element: does Expansion";
            this.vArr[3] = (Vertix) this.vArr[4].Clone();
      }
      else
      {
            optimize = true;
            optimizeElementsInVertix[whichParameterOfVertix] =
            "Element: does Reflection";
      }
}
else
{
   optimizeElementsInVertix[whichParameterOfVertix] =
   "Element: does nothing";
   this.vArr[3] = (Vertix) this.vArr[4].Clone();
```

```
}
if (this.vArr[3].Cost >= costTest)        // Contraction to C1
{
   this.vArr[3] = (Vertex) this.vArr[4].Clone();
   this.vArr[3] = Vertex.ContractToC1( this.vArr[0], this.vArr[1], this.vArr[2],
   this.vArr[3], whichParameterOfVertix);
   ComputeCost(3);
   if ( costTest <= this.vArr[3].Cost)     // Check C1
   {
      this.vArr[3] = (Vertex) this.vArr[4].Clone();
      this.vArr[3] = Vertex.ContractToC2( this.vArr[0], this.vArr[1], this.vArr[2],
      this.vArr[3], whichParameterOfVertix);
      ComputeCost(3);
      if (this.vArr[3].Cost < costTest)     // c2 is better than c1
      {
            optimize = true;
            optimizeElementsInVertix[whichParameterOfVertix] =
            "Element: does Contraction toward C2";
      }
      else
      {
            this.vArr[3] = (Vertex) this.vArr[4].Clone();
            optimizeElementsInVertix[whichParameterOfVertix] =
            "Element: does nothing";
      }
    }
```

```
    else
    {
            optimize = true;
            optimizeElementsInVertix[whichParameterOfVertix] =
            "Element: does Contraction toward C1";
    }
}
if (this.vArr[3].Cost >= costTest)
{
  this.vArr[3] = (Vertix) this.vArr[4].Clone();
  this.vArr[3] = Vertix.ShrinkGToB( this.vArr[0], this.vArr[1],
  this.vArr[3], whichParameterOfVertix);
  ComputeCost(3);
  if (costTest <= this.vArr[3].Cost)
  {
     this.vArr[3] = (Vertix) this.vArr[4].Clone();
     this.vArr[3] = Vertix.ShrinkWToB( this.vArr[0], this.vArr[1],
     this.vArr[3], whichParameterOfVertix);
     ComputeCost(3);
     if (this.vArr[3].Cost < costTest)
     {
            optimize = true;
            optimizeElementsInVertix[whichParameterOfVertix] =
            "Element: does Shrink Worse to Best";
     }
     else
```

```
            {
                this.vArr[3] = (Vertix) this.vArr[4].Clone();
                optimizeElementsInVertix[whichParameterOfVertix] =
                "Element: does nothing";
            }
        }
         else
         {
                optimize = true;
                optimizeElementsInVertix[whichParameterOfVertix] =
                "Element: does Shrink Good to Best";
         }
    }
     this.vArr[4].Cost = 0; this.vArr[4].A = -1.2 + (-1.2 * 0.05 * 2);
     this.vArr[4].B = 1.1; ComputeCost(4);
     return optimize;
}
public bool DoOptimizationWheenThStuck()
{
    bool optimize = false;
    double costTest = this.vArr[3].Cost;
    this.vArr[4] = Vertix.TryRefluction( this.vArr[0], this.vArr[1], this.vArr[2],
    this.vArr[4], 0);
    this.vArr[4] = Vertix.TryRefluction( this.vArr[0], this.vArr[1], this.vArr[2],
    this.vArr[4], 1);
    ComputeCost(4);
```

```
if (this.vArr[4].Cost < costTest)     // Try Expansion
{
   this.vArr[3] = Vertix.TryExpansion( this.vArr[0], this.vArr[1], this.vArr[2],
   this.vArr[3], 0);
   this.vArr[3] = Vertix.TryExpansion( this.vArr[0], this.vArr[1], this.vArr[2],
   this.vArr[3], 1);
   ComputeCost(3);
   if (this.vArr[3].Cost < this.vArr[4].Cost)     // Expansion is a better than Ref.
   {
        optimize = true;
        optimizeElementsInVertix[0] = "Element: does Expansion";
        optimizeElementsInVertix[1] = "Element: does Expansion";
   }
   else
   {
         optimize = true;
         this.vArr[3] = (Vertix) this.vArr[4].Clone();
         optimizeElementsInVertix[0] = "Element: does Reflection";
         optimizeElementsInVertix[1] = "Element: does Reflection";
    }
}
if (this.vArr[3].Cost >= costTest)     // contract toward either c1 or c2
{
   this.vArr[4] = Vertix.ContractToC1( this.vArr[0], this.vArr[1], this.vArr[2],
   this.vArr[4], 0);
   this.vArr[4] = Vertix.ContractToC1( this.vArr[0], this.vArr[1], this.vArr[2],
```

```
this.vArr[4], 1);
ComputeCost(4);
if ( costTest <= this.vArr[4].Cost)     // Check C1
{
   this.vArr[4] = Vertix.ContractToC2( this.vArr[0], this.vArr[1], this.vArr[2],
   this.vArr[4], 0);
   this.vArr[4] = Vertix.ContractToC2( this.vArr[0], this.vArr[1], this.vArr[2],
   this.vArr[4], 1);
   ComputeCost(4);
   if (this.vArr[3].Cost < costTest)     // c2 is better than c1
   {
      this.vArr[3] = (Vertix) this.vArr[4].Clone();
      optimize = true;
      optimizeElementsInVertix[0] = "Element: does Contraction toward C2";
      optimizeElementsInVertix[1] = "Element: does Contraction toward C2";
   }
   else
   {
       optimizeElementsInVertix[0] = "Element: does nothing";
       optimizeElementsInVertix[1] = "Element: does nothing";
   }
}
else
{
   this.vArr[3] = (Vertix) this.vArr[4].Clone();
   optimize = true;
```

```
            optimizeElementsInVertix[0] = "Element: does Contraction toward C1";
            optimizeElementsInVertix[1] = "Element: does Contraction toward C1";
        }
    }
    if (this.vArr[3].Cost >= costTest)
    {
        this.vArr[4] = Vertix.ShrinkGToB( this.vArr[0], this.vArr[1],
        this.vArr[4], 0);
        this.vArr[4] = Vertix.ShrinkGToB( this.vArr[0], this.vArr[1],
        this.vArr[4], 1);
        ComputeCost(4);
        if ( costTest <= this.vArr[4].Cost)     // Contract W to B
        {
            this.vArr[4] = Vertix.ShrinkWToB( this.vArr[2], this.vArr[0],
            this.vArr[4], 0);
            this.vArr[4] = Vertix.ShrinkWToB( this.vArr[2], this.vArr[0],
            this.vArr[4], 1);
            ComputeCost(4);
            if (this.vArr[4].Cost < costTest)
            {
                    optimize = true;
                    optimizeElementsInVertix[0] = "Element: does Shrink Worse to Best";
                    optimizeElementsInVertix[1] = "Element: does Shrink Worse to Best";
            }
            else
            {
```

```
                    optimizeElementsInVertix[0] = "Element: does nothing";
                    optimizeElementsInVertix[1] = "Element: does nothing";
                }
            }
            else
            {
                this.vArr[3] = (Vertix) this.vArr[4].Clone();
                optimize = true;
                optimizeElementsInVertix[0] = "Element: does Shrink Good to Best";
                optimizeElementsInVertix[1] = "Element: does Shrink Good to Best";
            }
        }
            this.vArr[4].Cost = 0; this.vArr[4].A = -1.2 + (-1.2 * 0.05 * 2);
            this.vArr[4].B = 1.1; ComputeCost(4);
            return optimize;
        }
    }
}
```

Finally, we need to create a function in the Form class to call the HNM algorithm as follows.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;
namespace RosenBrockFunction
{
        public partial class Form1 : Form
        {
          Simplex simp = new Simplex();
          private void RunHNM(int numSimplexEvaluateions)
          {
            for (int i =0; i < simp.VArr.Length; i++)
            {
              simp.ComputeCost(i);
            }
            Array.Sort(simp.VArr);
            for (int j =0; j < numSimplexEvaluateions; j++)
            {
              double whatCostFind = simp.VArr[3].Cost;
              for (int i =0; i < 2; i++)
              {
                bool didParameterOptimized =
                simp.DoOptimizationOnVertixElements(i);
              }
              if (whatCostFind <= simp.VArr[3].Cost &&
              simp.OptimizeElementsInVertix[0] == "Element: does nothing" &&
              simp.OptimizeElementsInVertix[1] == "Element: does nothing")
              {
```

```
                    bool didParameterOptimized =
                    simp.DoOptimizationWheenThStuck();
                }
                Array.Sort(simp.VArr);
            }
        }
    }
}
```